Research Workshop of the
Israel Science Foundation

Proceedings of the 3rd Workshop on

Planning and Robotics (PlanRob-15)

Edited By:

Alberto Finzi, Felix Ingrand, AndreA Orlandini

Jerusalem, Israel 7-8/6/2015

Organizing Commitee

Alberto Finzi
Federico II University, Naples, Italy

Felix Ingrand
LAAS-CNRS, Toulouse, France

AndreA Orlandini
National Research Council (CNR-ISTC), Rome, Italy

Program committee

Rachid Alami, LAAS-CNRS, France
Sara Bernardini, King's College, UK
Amedeo Cesta, CNR-ISTC, Italy
Marcelo Cirillo, Orebro University, Sweden
Alberto Finzi, Federico II University, Italy
Robert Fitch, University of Sydney, Australia
Maria Fox, King's College, UK
Malik Ghallab, LAAS-CNRS, France
Joachim Hertzberg, University of Osnabrueck, Germany
Felix Ingrand, LAAS-CNRS, France
Luca Iocchi, Sapienza University, Italy
Gal Kaminka, Bar Ilan University, Israel
Sven Koenig, University of Southern California, USA
Jonas Kvarnstrom, Linköpings University, Sweden
Daniele Magazzeni, King's College, UK
Daniele Nardi, Sapienza University, Italy
Goldie Nejat, Toronto University, Canada
AndreA Orlandini, CNR-ISTC, Italy
Federico Pecora, Orebro University, Sweden
Frederic Py, MBARI, USA
Maria Dolores Rodriguez Moreno, Alcala University, Spain
Enrico Scala, ANU Research School in Computer Science, Australia
Siddarth Srivastava, Berkeley University, USA
Florent Teichteil, Onera, France

Additional Reviewers:

Guglielmo Gemignani, Sapienza University, Italy
Uwe Köckemann, Orebro University, Sweden
Stefan Konecny, Orebro University, Sweden
Francesco Riccio, Sapienza University, Italy
Tansel Uras, University of South California, USA

Foreword

Robotics is one of the most appealing and natural applicative area for the Planning and Scheduling (P&S) research activity, however, this potential interest seems not reflected in an equally important research production for the robotics community. On the other hand, the fast development of field and social robotics applications poses planning as a central issue in the robotic research with several real-world challenges for the planning community.

In this perspective, the goal of the PlanRob workshop is twofold. From one side, it aims at providing a fresh impulse for the ICAPS community to recast its interests towards robotics problems and applications. On the other side, its goal is to attract representatives from the robotics community to discuss their challenges related to planning for autonomous robots as well as their expectations from the P&S community.

The workshop aims at constituting a stable, long-term establishment of a forum on relevant topics concerned with the interactions between Robotics and P&S communities presenting a stimulating environment where researchers could discuss about the opportunities and challenges for P&S when applied to Robotics.

Started during ICAPS 2013 in Rome (Italy) and followed by the second edition at ICAPS 2014 in Portsmouth (NH, USA), the PlanRob WS series (http://pst.istc.cnr.it/planrob/) has gathered very good feedback from the P&S community which is also confirmed by the organization of a specific Robotics Track at both ICAPS 2014 and ICAPS 2015 (this year chaired by Reid Simmons and Micheal Beetz). This third edition of the PlanRob workshop has been proposed again in synergy with this track to further enforce the original goal and to maintain a more informal forum where also more preliminary/visionary work can be discussed as well as more direct and open interactions/discussions may find the right place.

In our opinion, PlanRob'15 succeeded in achieving these objectives. Indeed, 17 papers have been accepted for oral presentation covering many relevant topics such as high-level task planning, task and motion planning, planning and execution for robots, multi-robot framework, human-robot interaction, benchmarking, etc. This seems to us a really good result for the workshop and, overall, it confirms a good feedback from the ICAPS community (but not only) on PlanRob topics.

Finally, two notable researchers have accepted our invitation to provide a keynote talk and complete an already rich and interesting program: Reid Simmons (Carnegie Mellon University - CMU) talking about multi-robot coordination and robust autonomy and Steve Chien (Jet Propulsion Laboratory NASA – JPL-NASA) talking about the use of constraint-based reasoning in the Rosetta mission.

Alberto Finzi, Felix Ingrand and AndreA Orlandini

The PlanRob'15 Chairs

PlanRob 2015 is partially supported by the FourByThree project (http://www.fourbythree.eu - EU H2020 G.A. FoF- 637095) and the SHERPA project (http://www.sherpa-project.eu EU FP7 G.A. ICT-600958).

Table of Contents

Human-Robot Interaction

Estimating the Probability of Meeting a Deadline in Hierarchical Plans

Liat Cohen and **Solomon Eyal Shimony** and **Gera Weiss**
Computer Science Department
Ben Gurion University of The Negev
Beer-Sheva, Israel 84105
{liati,shimony,geraw}@cs.bgu.ac.il

Abstract

Given a hierarchical plan with uncertain task times, or other resource consumption, it is frequently of interest to compute the probability that a given plan will satisfy a given deadline, or resource limits. We show that this problem is NP-hard, and provide a polynomial-time approximation algorithm for it. We also show that computing the expected time of a task network is NP-hard even though this problem is easy for both sequence nodes and for parallel nodes. We examine the approximation bounds empirically and demonstrate where our scheme is superior to sampling and to exact computation.

Introduction

Robotics systems typically have tasks that are described hierarchically, and which can run concurrently. A common approach to describe the tasks is by using a Hierarchical Task Network (HTN), and there exist planners that automatically operate on such descriptions and generate hierarchical plans (Erol, Hendler, and Nau 1994; Nau et al. 1998; 2003; Kelly, Botea, and Koenig 2008). The work on the last DARPA Robotics Challenge (DRC, www.darpa.mil/NewsEvents/Releases/2012/10/24.aspx) in the ROBIL team highlighted the need for evaluating the desirability of hierarchical plans in terms of resource consumption, such as fuel, cost, or time. Once computed, these values can be used to decide which of a set of plans, all of which are valid as far as achieving the goal(s) are concerned, is better given a user-specified utility function.

In the DRC and other semi-automated robotics tasks, this information can be used to support runtime monitoring of the resources. Then one can generate alerts to the execution software or human operator if resource consumption in practice is far from the expected value, or has a high probability of surpassing a given threshold. Our ROBIL team system had manually generated hierarchical plans. For example, Figure 1, is a plan from DRC task description for the (simulation phase) "pick-up" scenario. This plan describes the steps needed to be taken by the robot in order to pick up an object. In the figure, arrow-shaped boxes stand for sequential tasks, parallelograms for parallel tasks, and rectangles for primitive tasks. A scheme for monitoring was implemented using a simple sampling algorithm, which delivered answers on questions such as: "what is the probability that this plan will complete execution in 5 minutes?" in real-time. Due to the stochastic nature of the algorithm, the results were unstable, and the question of whether this can be done deterministically, the focus of this paper, was raised.

We assume here that a hierarchical plan is given, where the resource consumption of the primitive actions in the network is uncertain and provided as a probability distribution. The problem is to compute a property of interest of the distribution for the entire task network, such as expected resource consumption of the entire plan, or the probability that the entire plan's execution can be completed before a given deadline. While most tools aim at good average performance of the plan, in which case one may ignore the full distribution and consider only the expected resource consumption (Bonfietti, Lombardi, and Milano 2014), our paper focuses on providing guarantees for the probability of meeting deadlines. Since in the above-mentioned applications for these computations, one needs results in real-time (for monitoring) or multiple such computations (in comparing candidate plans), efficient computation here is crucial, and more important than in, e.g. off-line planning.

Task descriptions may contain various parameters and other complications (Erol, Hendler, and Nau 1994; Nau et al. 1998). For the sake of simplicity, we will assume that there is only one resource of interest, such as execution time; and that we are given a fully instantiated plan, abstracting away from the complex task descriptions. Distributions over the execution time of each primitive task are provided as input. We allow tasks to be run either in sequence, or in parallel (also called "concurrent" tasks (Gabaldon 2002)). Under the simplifying assumptions we make (see problem statement section), the problems we need to solve entail computing a representation of the distribution for a sum of random variables (due to the tasks in sequence), as well as that of a maximum of random variables (due to the tasks in parallel).

In this paper, we focus on the issue of probability of satisfying a deadline. We show that computing this probability is NP-hard even for the simple sum of independent random variables, the first contribution of this paper. A deterministic polynomial-time approximation scheme for this problem is proposed, the second contribution of this paper. Error bounds are analyzed and are shown to be tight. For discrete random variables with finite support, finding the distribution of the maximum can be done in low-order polynomial time. However, when compounded with errors generated due to

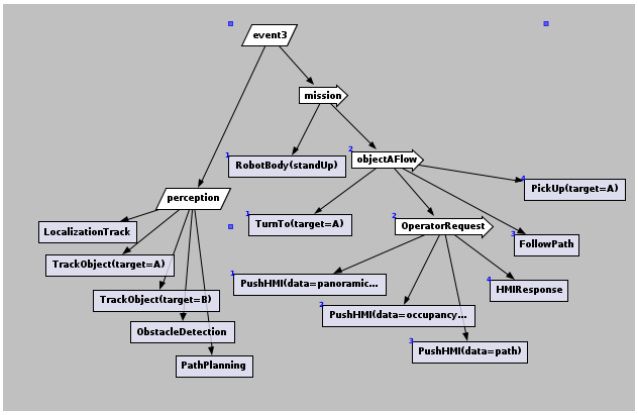


Figure 1: A plan for Pick-Up challenge task, 18 nodes

approximation in subtrees, handling this case requires careful analysis of the resulting error. The approximations developed for both sequence and parallel nodes are combined into an overall algorithm for task trees, with an analysis of the resulting error bounds, yielding a polynomial-time (additive error) approximation scheme for computing the probability of satisfying a deadline for the complete network, another contribution of this paper.

We also briefly consider computing *expected* completion time. Since for discrete random variables, in parallel nodes one can compute an exact distribution efficiently, it is easy to compute an *expected completion time* in this case as well as for sequence nodes. Despite that, we show that for trees with both parallel and sequence nodes, computing the expected completion time is NP-hard.

Finally, we provide some supporting experiments in order to examine the quality of approximation in practice when compared to the theoretical error bounds. A simple sampling scheme (Mainprice et al. 2011) is also provided as a yardstick, even though the sampling does not come with error guarantees, but only bounds in probability. This is done for randomly generated task networks, as well as some task networks taken from the ROBIL team task (http://in.bgu.ac.il/en/Pages/news/dar_pa.aspx) descriptions implementation for some of the DARPA robotics (simulation phase) challenge scenarios.

Problem statement

We are given a hierarchical plan represented as a task tree τ, consisting of three types of nodes: primitive actions as leaves, sequence nodes, and parallel nodes. Primitive action nodes contain distributions over their resource consumption. Although any other resource can be represented, we will assume henceforth in order to be more concrete that the only resource of interest is time. A sequence node v_s denotes a task that has been decomposed into subtasks, represented by the children of v_s, and which must be executed in sequence in order to execute v_s. We assume that a subtask of v_s begins as soon as its predecessor in the sequence terminates. Task v_s terminates when its last subtask terminates. A parallel node v_p also denotes a decomposed task, but here the subtasks begin execution in parallel immediately when task v_p begins execution; v_p terminates as soon as all of the children of v_p terminate.

Resource consumptions are uncertain, and are described as probability distributions in the leaf nodes. Although in principle we can allow statistical dependencies, we assume in this paper that these distributions are described by independent (but *not* identically distributed) random variables. We will also assume initially that the random variables are discrete and have finite support (i.e. the number of values for which the probability is non-zero is finite), although later on we will relax this assumption. As the resource of interest is assumed to be completion time, we assume that each leaf node v is associated with a completion-time distribution P_v, in some cases represented as a cumulative distribution function form (CDF) F_v.

The main computation problem that we tackle in this paper is: Given a task tree τ and a deadline T, compute the probability that τ will satisfy the deadline T (i.e. terminate in time not greater than T). We show that this problem is NP-hard and provide an approximation algorithm for it. The above *deadline problem* reflects a step utility function: a constant positive utility U for all t less than or equal to a deadline time T, and 0 for all $t > T$. We also briefly consider a linear utility function, requiring computation of the expected completion time of τ and show that this *expectation problem* is also NP-hard.

Sequence nodes

Since the completion time of a sequence node is the sum of the completion time of its components, the deadline problem on sequence nodes entails computation of (part of) the CDF of a sum of random variables. As the latter problem is NP-hard (shown later on), sequence nodes are the main challenge, and we thus begin with an approximation algorithm for sequence nodes.

The main ingredient in our approximation scheme is the Trim operator specified as follows:

Definition 1 (The Trim operator)**.** *For a discrete random variable X and a parameter $\varepsilon > 0$, consider the sequence of elements in the support of X defined recursively by: $x_1 = \min \mathrm{support}(X)$ and, if the set $\{x > x_i \colon Pr(x_i < X \leq x) > \varepsilon\}$ is not empty,*

$$x_{i+1} = \min\{x > x_i \colon Pr(x_i < X \leq x) > \varepsilon\}.$$

Let l be the length of this sequence, i.e., let l be the first index for which the set above is empty. For notational convenience, define $x_{l+1} = \infty$. Now, define $X' = \mathrm{Trim}(X, \varepsilon)$ to be the random variable specified by:

$$Pr(X'{=}x) := \begin{cases} Pr(x_i \leq X < x_{i+1}) & \text{if } x{=}x_i \in \{x_1, \ldots, x_l\}; \\ 0 & \text{otherwise.} \end{cases}$$

where $x_1, \ldots, x_l$ is the sequence defined above for the given ε.

For example, if X is a random variable such that $Pr(X{=}1){=}0.1, Pr(X{=}2){=}0.1$ and $Pr(X{=}4){=}0.8$, the random variable $X' {=} \mathrm{Trim}(X, 0.5)$ is given by

$Pr(X'{=}1){=}0.2$ and $Pr(X'{=}4){=}0.8$. Intuitively, the Trim operator removes consecutive domain values whose accumulated probability is less than ε and adds their probability mass to the element in the support that precedes them.

Using the Trim operator, we are now ready to introduce the main operator of this section:

Definition 2 (The Sequence operator)**.**

$$\text{Sequence}(X_1,\ldots,X_n,\varepsilon) := \text{Trim}(\text{Sequence}(X_1,\ldots,X_{n-1},\varepsilon)+X_n,\varepsilon)$$

and $\text{Sequence}(X_1,\varepsilon)=\text{Trim}(X_1,\varepsilon)$.

This operator takes a set of random variables and computes a random variable that represents an approximation of their sum by applying the Trim operator after adding each of the variables. The parameter ε, as shown later on, specifies the accuracy of approximation.

The Sequence operator can be implemented by the procedure outlined in Algorithm 1. In essence, the algorithm computes the distribution $\sum_{i=0}^{n} X_i$ using convolution (the Convolve() operator in line 3) in a straightforward manner. Computing the convolution in the case of discrete random variables is itself straightforward and not further discussed here. However, since the support of the resulting distribution may be exponential in the number of convolution operations, the algorithm must trim the distribution representation to avoid this exponential blow-up. This decreases the support size, while introducing error. The trick is to keep the support size under control, while making sure that the error does not increase beyond a desired tolerance. Note that the size of the support can also be decreased by simple "binning" schemes, but these do not provide the desired guarantees. In the algorithm, the PDF of a random variable X_i is represented by the list D_{X_i}, which consists of (x,p') pairs, where $x \in \text{support}(X_i)$ and p' is the probability $Pr(X_i{=}x)$, the latter denoted by $D_{X_i}[x]$ in the algorithm. We assume that D_{X_i} is always kept sorted in increasing order of x.

We proceed to show that Algorithm 1 indeed approximates the sum of the random variables, and analyze its accuracy/efficiency trade-off. A notion of approximation relevant to deadlines is as follows:

Definition 3. *For random variables X', X, and $0 \le \varepsilon \le 1$ we write $X' \approx_\varepsilon X$ if $0 \le Pr(X' \le T) - Pr(X \le T) \le \varepsilon$ for all $T > 0$.*

Note that this definition is asymmetric, because, as shown below, our algorithm for the deadline problem, never underestimates the exact value. We, of course, considered also the option to define the `Trim` operator such that the weight is shifted to the closest value (not always to the lower bound or always to the upper bound) or maybe to the expected value. This is a viable option, but it is not clear to us how to device guarantees with such an operator. Perhaps further research may reveal similar properties of other notions of approximations that apply to the symmetric operator.

For the proof of the next lemma we will establish the following technical claim:

Claim 1. *Let $(a_i)_{i=1}^n$ and $(b_i)_{i=1}^n$ be sequences of real numbers such that $\sum_{i=1}^{k} a_i \ge 0$ for all $1 \le k \le n$ and $b_1 \ge b_2 \ge \cdots \ge b_n \ge 0$. Then we have: $\sum_{i=1}^{n} a_i b_i \ge 0$*

Algorithm 1: `Sequence` $(X_1,\ldots,X_n,\varepsilon)$

```
1  D = ((0,1)) // Dummy random var.: 0 with prob. 1
2  for i = 1 to n do
3      D=Convolve(D, D_{X_i})
4      D=Trim(D, ε)
5  return D
6  Procedure Trim(D,ε)
7      D' = ()
8      d_0 = d_prev = min support(D)
9      p = 0
10     foreach d ∈ support(D) \ {d_0} in ascending order do
11         if p + D[d] ≤ ε then
12             p = p + D[d]
13         else
14             Append (d_prev, D[d_prev] + p) to D'
15             d_prev = d
16             p = 0
17     Append (d_prev, D[d_prev] + p) to D'
18     return D'
```

Proof. By induction on the length of the sequence, n. If $n = 1$ the claim is trivial. For $n > 1$,

$$\begin{aligned}\sum_{i=1}^{n} a_i b_i &= \sum_{i=1}^{n-1} a_i b_i + a_n b_n \\ &\ge \sum_{i=1}^{n-1} a_i b_i - \sum_{i=1}^{n-1} a_i b_n \\ &= \sum_{i=1}^{n-1} a_i (b_i - b_n).\end{aligned}$$

The sequence $b'_i = b_i - b_n$ satisfies the condition so, by the induction hypothesis, we get that $\sum_{i=1}^{n-1} a_i b'_i \ge 0$, and therefore, $\sum a_i b_i \ge 0$. □

The following lemma bounds the approximation error of sums of random variables. For technical reasons, we will focus from now on random variables with integer values. This restriction is only to simplify the analysis. One can reduce an instance of the deadline problem on general discrete random variables to a problem on integer random variables.

Lemma 1. *For discrete integer random variables X_1, X'_1, X_2, X'_2 and $\varepsilon_1, \varepsilon_2 \in [0,1]$, if $X'_1 \approx_{\varepsilon_1} X_1$ and $X'_2 \approx_{\varepsilon_2} X_2$ then $X'_1 + X'_2 \approx_{\varepsilon_1+\varepsilon_2} X_1 + X_2$.*

Proof.

$$
\begin{aligned}
&Pr(X_1' + X_2' \le T) - Pr(X_1 + X_2 \le T) \\
&= \sum_{j=1}^{T} Pr(X_1'=j) \underbrace{Pr(X_2' \le T-j)}_{\le Pr(X_2 \le T-j)+\varepsilon_2} - \sum_{j=1}^{T} Pr(X_1=j) Pr(X_2 \le T-j) \\
&\le \sum_{j=1}^{T} Pr(X_1'=j)(Pr(X_2 \le T-j)+\varepsilon_2) - Pr(X_1=j) Pr(X_2 \le T-j) \\
&= \sum_{j=1}^{T} (Pr(X_1'=j) - Pr(X_1=j)) \underbrace{Pr(X_2 \le T-j)}_{\in [0,1]} + \varepsilon_2 \sum_{j=1}^{T} Pr(X_1'=j) \\
&\le \underbrace{\sum_{j=1}^{T} (Pr(X_1'=j) - Pr(X_1=j))}_{\le \varepsilon_1} + \varepsilon_2 \underbrace{\sum_{j=1}^{T} Pr(X_1'=j)}_{\in [0,1]} \\
&\le \varepsilon_1 + \varepsilon_2.
\end{aligned}
$$

Finally, we show that the difference between $Pr(X_1' + X_2' \le T)$ and $Pr(X_1 + X_2 \le T)$ is nonnegative:

$$
\begin{aligned}
&Pr(X_1' + X_2' \le T) - Pr(X_1 + X_2 \le T) \\
&= \sum_{j=1}^{T} Pr(X_1'=j) Pr(X_2' \le T-j) - \sum_{j=1}^{T} Pr(X_1=j) Pr(X_2 \le T-j) \\
&= \sum_{j=1}^{T} (Pr(X_1'=j) - Pr(X_1=j)) Pr(X_2 \le T-j) \\
&\qquad + \sum_{j=1}^{T} Pr(X_1'=j)(Pr(X_2' \le T-j) - Pr(X_2 \le T-j))
\end{aligned}
$$

The first term in this expression is non-negative by Claim 1 and the second is nonnegative because it is a sum of nonnegative numbers. □

The next lemma shows that $\texttt{Trim}(X, \varepsilon)$ is an ε-approximation of X.

Lemma 2. $\texttt{Trim}(X, \varepsilon) \approx_\varepsilon X$

Proof. Let $X' = \texttt{Trim}(X, \varepsilon)$. Let $x_1 < \cdots < x_m$ be the support of X' and let $l = \max\{i \colon x_i \le T\}$. We have,

$$Pr(X' = x_i) = Pr(x_i \le X < x_{i+1}) \tag{1}$$

because, after Trim, the probabilities of elements that were removed from the support are assigned to the element that precedes them. From Equation (1) we get:

$$
\begin{aligned}
&Pr(X' \le T) - Pr(X \le T) \\
&= \sum_{i=0}^{l-1} (Pr(X'=x_i) - Pr(x_i \le X < x_{i+1})) \\
&\qquad + (Pr(X'=x_l) - Pr(x_l \le X \le T)) \\
&= Pr(X'=x_l) - (Pr(x_l \le X < x_{l+1}) - Pr(T < X < x_{l+1})) \\
&= Pr(T < X < x_{l+1}) \in (0, \varepsilon]
\end{aligned}
$$

The inequality $Pr(T < X < x_{l+1}) \le \varepsilon$ follows from the observation that, for all i, $Pr(x_i < X < x_{i+1}) < \varepsilon$, because p is never greater than ε in Algorithm 1. □

For bounding the amount of memory needed for an implementation of our approximation algorithm, we also, in the next lemma, bound the size of the support of the trimmed random variable. The size of the support is a key factor because it reflects the size of the representation of the random variable.

Lemma 3. $|\operatorname{support}(\texttt{Trim}(X, \varepsilon))| \le 1/\varepsilon$

Proof. Let $X' = \texttt{Trim}(X, \varepsilon)$ and let $x_1 < \cdots < x_m$ be the support of X'. And, for notational convenience, let $x_{m+1} = \infty$. Let $p_i = \sum_{x_i < x < x_{i+1}} Pr(X=x)$. Then, $1 = \sum_{i=1}^{m} Pr(X'=x_i) = Pr(X=x_1) + \sum_{i=1}^{m-1}(p_i + Pr(X=x_{i+1})) + p_m$. According to algorithm 1, lines 11-12, $p_i \le \varepsilon$ and $p_i + Pr(X=x_{i+1}) > \varepsilon$ for all $0 \le i < m$. Therefore, $1 = \sum_{i=1}^{m} Pr(X'=x_i) > Pr(X'=x_1) + (m-1) \cdot \varepsilon + p_m$. Using the fact $Pr(X'=x_1) > 0$, we get: $(m-1) \cdot \varepsilon < 1$, therefore $m \le 1/\varepsilon$. □

The last two lemmas highlight the main idea behind our approximation algorithm: the Trim operator trades off approximation error for a reduced size of the support. The fact that this trade-off is linear allows us to get a linear approximation error in polynomial time, as shown in the following two theorems:

Theorem 1. *If $X_i' \approx_{\varepsilon_i} X_i$ for all $i \in \{1, \ldots, n\}$ and $\hat{X} = Sequence(X_1', \ldots, X_n', \varepsilon)$ then $\hat{X} \approx_e \sum_{i=1}^{n} X_i$, where $e = \sum_{i=1}^{n} \varepsilon_i + n\varepsilon$.*

Proof. (outline) For n iterations, according to Lemma 1, we get an accumulated error of $\varepsilon_1 + \cdots + \varepsilon_n$. In addition, from Lemma 2, we get an additional error of at most $n\varepsilon$ due to the trimming process. □

Theorem 2. *Assuming that $m \le 1/\varepsilon$, the procedure $Sequence(X_1', \ldots, X_n', \varepsilon)$ can be computed in time $O((nm/\varepsilon)\log(m/\varepsilon))$ using $O(m/\varepsilon)$ memory, where m is the size of the largest support of any of the X_i's.*

Proof. From Lemma 3, the size of the list D in Algorithm 1 is at most m/ε immediately after the convolution, after which it is trimmed, and thus the space complexity is $O(m/\varepsilon)$. The operator Convolve thus takes time $O((m/\varepsilon)\log(m/\varepsilon))$, where the logarithmic factor is required internally for sorting. Since the runtime of the Trim operator is linear, and the outer loop iterates n times, the overall run-time of the algorithm is $O((nm/\varepsilon)\log(m/\varepsilon))$. □

To show that the error bound provided in Theorem 1 is tight, we demonstrate by the following example that there are n random variables $X_1, \ldots, X_n$ for which $Sequence(X_1, \ldots, X_n, \varepsilon/n)$ indeed results in error ε.

Example 1. *Let $0 \le \varepsilon < 1$ and $n \in \mathbb{N}$ such that $1 - \varepsilon > \varepsilon/n$, i.e., ε is small or n is large. Consider, for $\delta > 0$ that we will choose to be very small, the random variable X_1 defined by*

$$
Pr(X_1=x) = \begin{cases}
\delta & x = 0, \\
\varepsilon/(n(1-\delta)^x) & x \in \{1, \ldots, n\}, \\
1 - \delta - \sum_{x=1}^{n} \frac{\varepsilon}{n(1-\delta)^x} & x = n+1, \\
0 & \textit{otherwise}
\end{cases}
$$

and, for $i \in \{2,\ldots,n\}$, let the random variables X_i be:

$$Pr(X_i{=}x) = \begin{cases} 1-\delta & x=0, \\ \delta & x=n^2, \\ 0 & \textit{otherwise} \end{cases}$$

The distribution of $X = X_1 + X_2$ is

$$Pr(X{=}x){=}\begin{cases} \delta(1-\delta) & x=0, \\ \varepsilon/n & x=1, \\ \varepsilon/(n(1-\delta)^{x-1}) & x\in\{2,\ldots,n\}, \\ (1{-}\delta)Pr(X_1{=}n{+}1) & x=n+1, \\ \delta Pr(X_1{=}x{-}n^2) & n^2{\le}x{\le}n^2{+}n{+}1 \\ 0 & \textit{otherwise} \end{cases}$$

The idea here is that the convolution with X_2 results in a random variable that is similar in "shape" to X_1, if we ignore numbers that tend to zero as δ approaches zero. The convolution also modifies the probability $Pr(X{=}1)$ from slightly greater than ε/n to precisely ε/n, which will then allow it to be trimmed.

Then, if we apply $\texttt{Trim}(X_1 + X_2, \varepsilon/n)$, when δ is sufficiently small, we get the random variable X' whose probability distribution is:

$$Pr(X'{=}x){=}\begin{cases} \delta(1-\delta)+\varepsilon/n & x=0, \\ \varepsilon/(n(1-\delta)^{x-1}) & x\in\{2,\ldots,n\}, \\ 1-Pr(X'{<}n+1) & x=n+1, \\ 0 & \textit{otherwise.} \end{cases}$$

Note that indeed trimming shifts the mass from $Pr(X = 1) = \varepsilon/n$ to $Pr(X' = 0)$. This repeats in all steps so, after n steps, we get a random variable X' such that $Pr(X'{=}0) \xrightarrow{\delta\to 0} \varepsilon$. Therefore, $Pr(Sequence(X_1,\ldots,X_n,\varepsilon/n){\le}0){-}Pr(X_1{+}\ldots{+}X_n{\le}0)$ approaches ε as δ approaches zero which means that there exist no $\varepsilon'{<}\varepsilon$ such that $Sequence(X_1,\ldots,X_n,\varepsilon/n) \approx_{\varepsilon'} X_1 + \cdots + X_n$ for all $\delta > 0$.

Parallel nodes

Unlike sequence composition, the deadline problem for parallel composition is easy to compute, since the execution time of a parallel composition is the maximum of the durations of the subtasks. Thus we have:

$$\begin{aligned} &Pr(\max\{X_1,\ldots,X_n\} \le T) = \\ &= Pr(X_1 \le T \wedge \cdots \wedge X_n \le T) = \prod_{i=1}^{n} Pr(X_i \le T) \quad (2) \end{aligned}$$

where the last equality follows from independence of the random variables. We denote the construction of the CDF using Equation (2) by $\texttt{Parallel}(X_1,\ldots,X_n)$. If the random variables are all discrete with finite support, $\texttt{Parallel}(X_1,\ldots,X_n)$ incurs space linear in the size of the input, and the computation time is $O(nmlog(n))$.

If the task tree consists only of parallel nodes, one can compute the exact CDF, with the same overall runtime. However, when the task tree contain both sequence and parallel nodes we may get only approximate CDFs as input, and now the above straightforward computation can compound the errors. When the input CDFs are themselves approximations, we bound the resulting error by the following lemma.

Lemma 4. *For discrete random variables $X'_1,\ldots X'_n$, $X_1,\ldots,X_n$, if for all $i = 1,\ldots,n$, $X'_i \approx_{\varepsilon_i} X_i$ and $0 \le \varepsilon_i \le \frac{1}{n(Kn+1)}$ for $K > 0$, then, for any $\varepsilon \ge \varepsilon_i$, we have: $\max\{X'_1,\ldots,X'_n\} \approx_e \max\{X_1,\ldots,X_n\}$ where $e = \sum_{i=1}^{n} \varepsilon_i + \varepsilon/K$.*

Proof.

$$\begin{aligned} &Pr(\max\{X'_1,\ldots,X'_n\}{\le}T) - Pr(\max\{X_1,\ldots,X_n\}{\le}T) \\ &= \prod_{i=1}^{n} Pr(X'_i \le T) - \prod_{i=1}^{n} Pr(X_i \le T) \\ &\le \prod_{i=1}^{n}(Pr(X_i \le T) + \varepsilon_i) - \prod_{i=1}^{n} Pr(X_i \le T) \\ &\le \prod_{i=1}^{n}(1+\varepsilon_i) - 1 \le 1 + \sum_{i=1}^{n}\varepsilon_i + \sum_{k=2}^{n}\binom{n}{k}\varepsilon^k - 1 \\ &\le \sum_{i=1}^{n}\varepsilon_i + \underbrace{\sum_{k=2}^{n} n^k\varepsilon^k}_{\text{sum of a geo. series}} \le \sum_{i=1}^{n}\varepsilon_i + \frac{n^2\varepsilon^2}{1-n\varepsilon} \\ &\le \sum_{i=1}^{n}\varepsilon_i + \varepsilon/K \end{aligned}$$

Since $Pr(X'_i \le T) > Pr(X_i \le T)$ for each i, this expression is nonnegative. □

Task networks: mixed sequence and parallel

Given a task tree τ and an accuracy requirement $0 < \varepsilon < 1$, we generate a distribution for a random variable X'_τ which represents an approximation of the random variable X_τ which represents the duration for the entire task tree. We introduce the algorithm and prove that the algorithm indeed returns an ε-approximation of the completion time of the plan. For a node v, let τ_v be the sub tree with v as root and let $child_v$ be the set of children of v. We use the notation $|\tau|$ to denote the total number of nodes in τ.

Note that given a task tree τ, we can assume, without loss of generality, that there are no parallel nodes as children of parallel nodes, and no sequence nodes as children of sequence nodes, otherwise, we can easily merge them.

Algorithm 2 is a straightforward postorder traversal of the tree τ. The only non-trivial issue is handling the error as a "budget", in an amortized approach, as seen in the proof of the following theorem.

Theorem 3. *Given a task tree τ, let X_τ be a random variable representing the true distribution of the completion time for the network. Then $\texttt{Network}(\tau,\varepsilon) \approx_\varepsilon X_\tau$.*

Proof. We prove by (strong) induction on the size of τ that $\texttt{Network}(\tau,\varepsilon) \approx_\varepsilon X_\tau$. Induction base: $|\tau| = 1$, the node

Algorithm 2: `Network` (τ, ε)

Let v be the root of τ // Hence, $\tau_v = \tau$
$n_v = |child_v|$
if v *is a* Primitive *node* **then**
  return the distribution of v
if v *is a* Sequence *node* **then**
  for *all* $c \in child_v$ **do**
    X'_{τ_c} = `Network` $(\tau_c, \frac{|\tau_c|\varepsilon}{|\tau_v|})$
  return `Sequence` $(\{X'_{\tau_c}\}_{c \in child_v}, \frac{\varepsilon}{n_v|\tau_v|})$
if v *is a* Parallel *node* **then**
  for *all* $c \in child_v$ **do**
    X'_{τ_c} = `Network` $(\tau_c, \min(\frac{|\tau_c|\varepsilon}{|\tau_v|}, \frac{1}{n_v(|\tau_v|n_v+1)}))$
  return `Parallel` $(\{X'_{\tau_c}\}_{c \in child_v})$

must be primitive, and `Network` will just return the distribution unchanged which is obviously an ε-approximation of itself. Suppose the claim is true for $1 \leq |\tau| < n$. Let τ be a task tree of size n and let v be the root of τ. If v is a Sequence node, then for each child c of v we call Network recursively to compute a $|\tau_c|\varepsilon/|\tau_v|$-approximation. By the induction hypothesis that $X'_{\tau_c} \approx_{|\tau_c|\varepsilon/|\tau_v|} X_{\tau_c}$, and by Theorem 1, we get a maximum accumulated error of $\sum_{c \in child_v} |\tau_c|\varepsilon/|\tau_v| + \varepsilon/|\tau_v| = (n-1)\varepsilon/|\tau_v| + \varepsilon/|\tau_v| = \varepsilon$ for v, therefore, $\texttt{Sequence}(\{X'_{\tau_c}\}_{c \in child_v}, \varepsilon/n) \approx_\varepsilon X_\tau$ as required. If v is a Parallel node, then for each child c of v we call Network recursively to compute a e_c-approximation, where $e_c = \min(\frac{|\tau_c|\varepsilon}{|\tau_v|}, \frac{1}{n_v(|\tau_v|n_v+1)})$. We have, by the induction hypothesis, that $X'_{\tau_c} \approx_{e_c} X_{\tau_c}$, so $\sum_{c \in child_v} e_c \leq \sum_{c \in child_v} \frac{|\tau_c|\varepsilon}{|\tau_v|} \leq \varepsilon - \varepsilon/|\tau_v|$. Then, by Lemma 4, using $K = |\tau_v|$ and $n = n_v$, we get that $\texttt{Parallel}(\{X'_{\tau_c}\}_{c \in child_v}) \approx_\varepsilon X_\tau$ as required. □

Theorem 4. *Let N be the size of the task tree τ, and M the size of the maximal support of each of the primitive tasks. If $0 \leq \varepsilon \leq \frac{1}{N(N^2+1)}$ and $M < N/\varepsilon$, the run-time of the* `Network` *approximation algorithm is $O((N^5/\varepsilon^2)\log(N^3/\varepsilon^2))$. The algorithm uses $O(N^3/\varepsilon^2)$ memory.*

Proof. The run-time and space bounds can be derived from the bounds on `Sequence` and on `Parallel`, as follows. In the `Network` algorithm, the trim accuracy parameter is less than or equal to ε/N. The support size (called m in Theorem 2) of the variables input to `Sequence` are $O(N^2/\varepsilon)$. Therefore, the complexity of the `Sequence` algorithm is $O((N^4/\varepsilon^2)\log(N^3/\varepsilon^2))$ and the complexity of the `Parallel` operator is $O((N^3/\varepsilon)\log(N))$. The time and space for sequence dominate, so the total time complexity is N times the complexity of `Sequence` and the space complexity is that of `Sequence`. □

If the constraining assumptions on M and ε in Theorem 4 are lifted, the complexity is still polynomial: replace one instance of $1/\varepsilon$ by $max(m, 1/\varepsilon)$, and the other by $max(1/\varepsilon, N(N^2+1))$ in the runtime complexity expression.

Complexity results

We show that the deadline problem is NP-hard, even for a task tree consisting only of primitive tasks and one sequence node. Observe that such simple trees are actually *linear plans*. We begin by showing the following lemma on sums of random variables.

Lemma 5. *Let $Y = \{Y_1, \ldots, Y_n\}$ be a set of discrete real-valued random variables specified by probability mass functions with finite supports, $T \in \mathbb{Z}$, and $p \in [0,1]$. Then, deciding whether $Pr(\sum_{i=0}^n Y_i < T) > p$ is NP-Hard.*

Proof. : by reduction from *SubsetSum* (Garey and Johnson 1990, problem number SP13). Recall that *SubsetSum* is: given a set $S = \{s_1, \ldots, s_n\}$ of integers, and an integer target value T, is there a subset of S whose sum is exactly T? Given an instance of *SubsetSum*, create the two-valued random variables $Y_1, \ldots, Y_n$ with $Pr(Y_i = s_i) = 1/2$ and $Pr(Y_i = 0) = 1/2$. By construction, there exists a subset of S summing to T if and only if $Pr(\sum_{i=0}^n Y_i = T) > 0$.

Suppose we have an algorithm $A(Y, T, p)$ that can decide $Pr(\sum_{i=0}^n Y_i < T) > p$ in polynomial-time. Then, since the random variables Y_i are two-valued uniform random variables, the only possible values of p are integer multiples of $1/2^n$, and we can compute $p = Pr(\sum_{i=0}^n Y_i < T)$ using a binary search on p using n calls to A. To determine whether $Pr(\sum_{i=0}^n Y_i = T) > 0$, simply use this scheme twice, since $Pr(\sum_{i=0}^n Y_i = T) > 0$ is true if and only if $Pr(\sum_{i=0}^n Y_i < T) < Pr(\sum_{i=0}^n Y_i < T+1)$. □

Theorem 5. *Finding the probability that a task tree τ satisfies a deadline T is NP-hard.*

Proof. Consider a linear plan, i.e. a tree consisting of just leaf nodes, all being children of a single sequence node. The completion time of a sequence node is the sum of the completion times of its children. Therefore, the theorem follows immediately from Lemma 5. □

Finally, we consider the linear utility function, i.e. the problem of computing an expected completion time of a task network. Note that although for linear plans the *deadline problem* is NP-hard, the *expectation problem* is trivial because the expectation of the sum of random variables X_i is equal to the sum of the expectations of the X_is. For *parallel nodes*, it is easy to compute the CDF and therefore also easy to compute the expected value. Despite that, for task networks consisting of *both* sequence nodes and parallel nodes, these methods cannot be effectively combined, and in fact, we have:

Theorem 6. *Computing the expected completion time of a task network is NP-hard.*

Proof. By reduction from subset sum. Construct random variables ("primitive tasks") Y_i as in the proof of Lemma 5, and denote by X the random variable $\sum_{i=1}^n Y_i$. Construct one parallel node with two children, one being a sequence

node having the completion time distribution defined by X, the other being a primitive task that has a completion time T_j with probability 1. (We will use more than one such case, which differ only in the value of T_j, hence the subscript j). Denote by M_j the random variable that represents the completion time distribution of the parallel node, using this construction, with the respective T_j. Now consider computing the expectation of the M_j for the following cases: $T_1 = T + 1/2$ and $T_2 = T + 1/4$. Thus we have, for $j \in \{1, 2\}$, by construction and the definition of expectation:

$$\begin{aligned} E[M_i] &= T_j Pr(X \leq T_j) + \sum_{x > T_j} x\ Pr(X = x) \\ &= T_j Pr(X \leq T) + \sum_{x \geq T+1} x\ Pr(X = x) \end{aligned}$$

where the second equality follows from the Y_i all being integer-valued random variables (and therefore X is also integer valued). Subtracting these expectations, we have that the difference $E[M_1] - E[M_2]$ is

$$(T + \frac{1}{2})Pr(X \leq T) + \sum_{x \geq T+1} xPr(X = x) -$$
$$(T + \frac{1}{4})Pr(X \leq T) + \sum_{x \geq T+1} xPr(X = x)$$

which reduces to $\frac{1}{4}P(X \leq T)$. Therefore, using the computed expected values, we can compute $P(X \leq T)$ in polynomial time. Using two such computations, we can also compute $P(X = T)$, and the latter is non-zero if and only if there is a subset of S summing to T. □

Empirical Evaluation

We examine our approximation algorithm empirically in order to shed light on the approximation bounds in practice. We used the same machine for all experiments, Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz, and implemented all algorithms in the same programming language, Python. A summary of the results is presented in graphs and in tables. We also compare the results to exact computation of the CDF and to a simple stochastic sampling scheme. We used the following sampling algorithm, Algorithm 3, which traverse the task tree τ. In every experiment, this algorithm was executed s times, according to the number of samples where $s \in \{10^3, 10^4, 10^5, 10^6, 10^7\}$.

Four types of task trees are used in this evaluation: task trees used as execution plans for our ROBIL team entry in the DARPA robotics challenge (DRC simulation phase), linear plans (seq), plans for the Logistics domain (from IPC2 http://ipc.icaps-conference.org/), and randomly generated task trees. The primitive task distributions were uniform distributions discretized to M values. The plans from the DRC are shown in Figures 1, 2 and 3. For every entry of M in tables 1 and 2 the first line is the runtime in seconds, the second line presents the estimation error.

In the Logistics domain, packages are to be transported by trucks or airplanes. Hierarchical plans were generated by

Algorithm 3: SampleTree(τ)

```
Let v be the root of τ
if v is a Primitive node then
    return randomly choose duration according to the distribution of v
if v is a Sequence node then
    for c ∈ child(v) do
        X += SampleTree(τ_c)
if v is a Parallel node then
    for c ∈ child(v) do
        X = max(X, SampleTree(τ_c))
return X
```

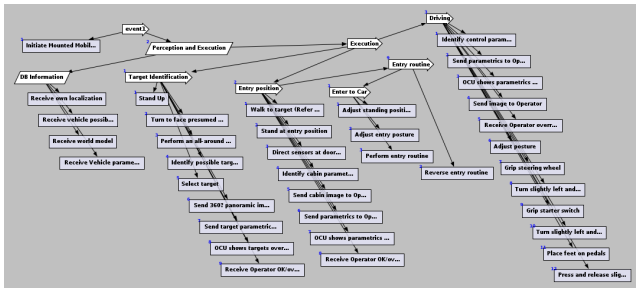


Figure 2: A plan for Drive challenge task, 47 nodes

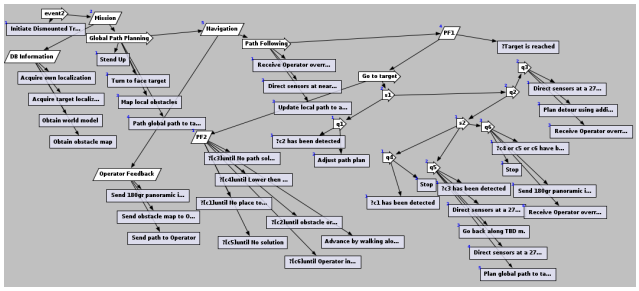


Figure 3: A plan for Walk challenge task, 57 nodes

the JSHOP2 planner (Nau et al. 2003) for this domain and consisted of one parallel node (packages delivered in parallel), with children all being sequential plans. The duration distribution of all primitive tasks is uniform. The support parameters were determined by the type of the task: in some tasks the distribution is fixed (such as for load and unload), and in others the distribution depends on the velocity of the vehicle and on the distance to be travelled. After running our approximation algorithm we actually also ran a variant that uses an inverted version of the `Trim` operator, providing a *lower* bound of the CDF, as well as the upper bound generated by Algorithm 2[1]. Running both variants allows us to bound the actual error, costing only a doubling of the runtime. Despite the fact that our error bound is theoretically

[1]Except for the randomly generated plans where we ran only Algorithm 2 and provided only the upper error bound.

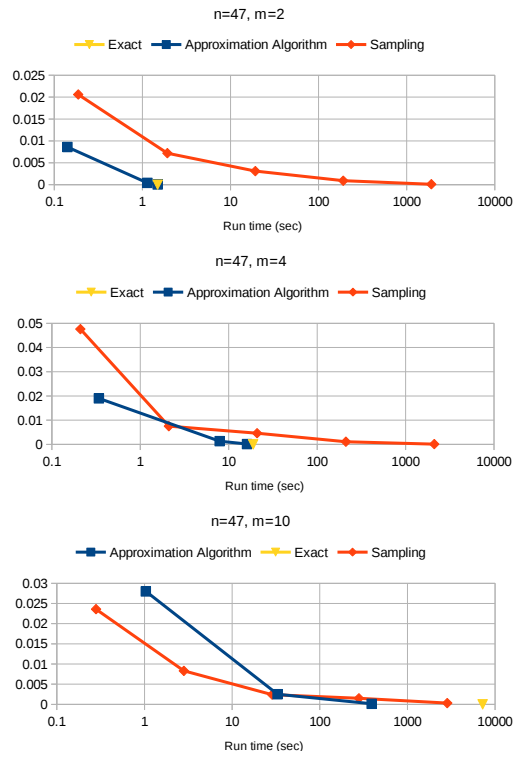


Figure 4: Execution times and estimation errors for the Drive plan (Figure 2).

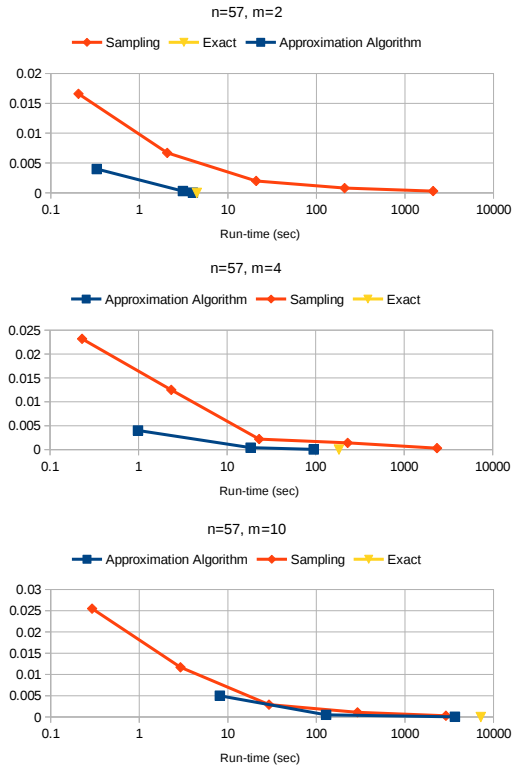


Figure 5: Execution times and estimation errors for the Walk plan (Figure 3).

tight, in practice and with actual distributions, according to tables 1 and 2, the resulting error in the algorithm is usually much better than the theoretical bound. The actual accuracy was sometimes as much as a factor of 20 better than the theoretical guarantee provided by ε.

Results for the various task trees are shown in tables 1 and 2. Errors are the maximum error in the CDF, measured from the true result when available, and from the bounds generated by the approximation algorithm using $\varepsilon = 0.0001$ when the exact algorithm timed out (over 2 hours). The exact algorithm times out in many cases when the number of tasks is 20 or more, except when size of the support M is very small, in which case it handles some more nodes, but still cannot handle 50 tasks even for $M = 2$. Both our approximation algorithm and the sampling algorithm handle all these cases, as our algorithm's runtime is polynomial in N, M, and $1/\varepsilon$ as is the sampling algorithm's (time linear in number of samples).

The advantage of the approximation algorithm is clearly the fact that it provides bound with certainty as opposed to the bounds in-probability provided by sampling. In addition, as predicted by theory, it is evident from the tables that the accuracy of the approximation algorithm improves linearly with $1/\varepsilon$ (and almost linearly with runtime in practice), whereas the accuracy of sampling improves only as a square root of the number of samples. Therefore, even in the cases where sampling initially outperformed the approxima-

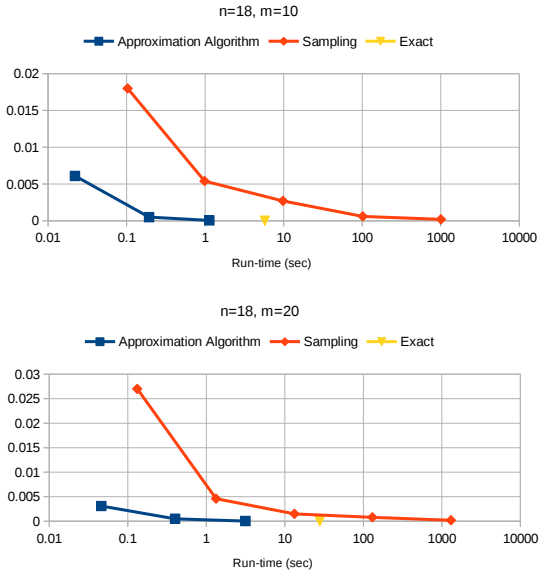


Figure 6: Execution times and estimation errors for the Pick Up plan (Figure 1).

Task Tree	M	Exact	Approximation algorithm			Sampling algorithm (s is # of samples)				
			ε=0.1	ε=0.01	ε=0.001	s=10^3	s=10^4	s=10^5	s=10^6	s=10^7
Drive N=47	2	1.49	0.141	1.14	1.49	0.187	1.92	19.11	190.4	1905
		0	[-0.0052, 0.0086]	[-0.0004, 0.0004]	[$-3.2\cdot10^{-5}$, $3.4\cdot10^{-5}$]	0.0206	0.0072	0.0031	0.0009	0.0001
	4	18.9	0.34	7.91	16.11	0.21	2.1	20.95	211.5	2113.6
		0	[-0.0096, 0.019]	[-0.0009, 0.0013]	[$-9.2\cdot10^{-5}$, $13\cdot10^{-5}$]	0.0476	0.0075	0.0046	0.0011	0.0001
	10	$>$ 2h	1.036	32.94	390.5	0.28	2.81	28.6	279.1	2844.4
		0	[-0.014, 0.028]	[-0.0014, 0.0025]	[$-9.5\cdot10^{-5}$, $14\cdot10^{-5}$]	0.0236	0.0083	0.0024	0.0015	0.0003
Walk N=57	2	4.46	0.33	3.1	4.03	0.205	2.06	20.86	208.1	2082.7
		0	[-0.0039, 0.004]	[-0.0003, 0.0003]	[$-3.1\cdot10^{-5}$, $3.2\cdot10^{-5}$]	0.0166	0.0067	0.002	0.0008	0.0003
	4	183.5	0.983	18.42	95.11	0.23	2.34	23.03	230.4	2352.4
		0	[-0.0038, 0.004]	[-0.0004, 0.0004]	[$-3.6\cdot10^{-5}$, $3.9\cdot10^{-5}$]	0.0232	0.0125	0.0022	0.0014	0.0003
	10	$>$ 2h	8.13	128.99	3668.2	0.293	2.92	29.16	291.3	2902.7
		0	[-0.0047, 0.0049]	[-0.0004, 0.0005]	[$-3.8\cdot10^{-5}$, $4\cdot10^{-5}$]	0.0255	0.0117	0.0029	0.0011	0.0003
Pick Up N=18	10	5.76	0.022	0.193	1.133	0.103	0.983	9.8	101.9	1006.8
		0	[-0.0041, 0.0061]	[-0.0003, 0.0005]	[$-3.5\cdot10^{-5}$, $5.8\cdot10^{-5}$]	0.018	0.0054	0.0027	0.0006	0.0002
	20	27.88	0.046	0.4	3.15	0.132	1.33	13.25	130.4	1305.9
		0	[-0.0038, 0.0031]	[-0.0006, 0.0005]	[$-3\cdot10^{-5}$, $3.5\cdot10^{-5}$]	0.027	0.0046	0.0015	0.0008	0.0002
Logistics1 N=34	2	0.014	0.007	0.009	0.009	0.239	2.03	19.3	193.9	1767
		0	[-0.0019, 0.0019]	0	0	0.0168	0.007	0.001	0.0009	0.0002
	4	22.98	0.048	1.3	13.1	0.2	2	20	205	1928
		0	[-0.0068, 0.0068]	[-0.0006, 0.0006]	[$-3.4\cdot10^{-5}$, $3.8\cdot10^{-5}$]	0.025	0.0057	0.0032	0.0005	0.0003
	10	$>$ 4h	0.25	8.26	475	0.26	2.64	26.4	267	2649
		0	[-0.008, 0.007]	[-0.0009, 0.0007]	0	0.018	0.011	0.003	0.0009	0.0004
Logistics2 N=45	2	0.07	0.02	0.06	0.06	0.23	2.35	23.4	234.7	2196
		0	[-0.002, 0.002]	0	0	0.013	0.015	0.004	0.001	0.0003
	4	373.3	0.2	7	82.9	0.25	2.5	25.6	256	2393
		0	[-0.004, 0.004]	[-0.0004, 0.0004]	[$-3.3\cdot10^{-5}$, $3.4\cdot10^{-5}$]	0.036	0.008	0.002	0.0006	0.0002
	10	$>$ 4h	2.19	120	6101	0.31	3.12	31.3	314	3139
		0	[-0.005, 0.006]	[-0.0004, 0.0006]	0	0.03	0.013	0.002	0.001	0.0002

Table 1: Runtime and errors estimation comparison (run time in seconds)

Task Tree	M	Exact	Approximation algorithm			Sampling algorithm		
			ε=0.1	ε=0.01	ε=0.001	s=10^3	s=10^4	s=10^5
Seq 10 N=10	4	0.23	0.003	0.02	0.148	0.054	0.545	5.336
		0	[-0.027, 0.041]	[-0.0027, 0.0041]	[$-2.2\cdot10^{-4}$, $2.5\cdot10^{-4}$]	0.0224	0.008	0.0017
	10	10.22	0.008	0.073	0.692	0.071	0.724	7.18
		0	[-0.0316, 0.0615]	[0.0033, 0.0067]	[$-2.6\cdot10^{-4}$, $5.2\cdot10^{-4}$]	0.027	0.0117	0.0038
Seq 20 N=20	2	0.23	0.003	0.02	0.285	0.054	0.545	9.62
		0	[-0.02, 0.0373]	[-0.0015, 0.0026]	[$-1.6\cdot10^{-4}$, $2.6\cdot10^{-4}$]	0.0266	0.0077	0.003
	4	$>$ 2h	0.011	0.106	1.208	0.105	1.066	10.74
		0	[-0.026, 0.025]	[-0.0025, 0.0025]	[$-2.7\cdot10^{-4}$, $2.3\cdot10^{-4}$]	0.039	0.01	0.002
	10	$>$ 2h	0.035	0.331	4.67	0.145	1.473	14.38
		0	[-0.027, 0.027]	[-0.0028, 0.0027]	[$-3\cdot10^{-4}$, $2.5\cdot10^{-4}$]	0.032	0.007	0.0042
Seq 50 N=50	2	$>$ 2h	0.028	0.28	3.593	0.236	2.366	24.71
		0	[-0.032, 0.032]	[-0.0028, 0.0028]	[$-2.8\cdot10^{-4}$, $2.4\cdot10^{-4}$]	0.0193	0.007	0.0024
	4	$>$ 2h	0.079	0.81	11.145	0.265	2.68	26.84
		0	[-0.035, 0.035]	[-0.0036, 0.0035]	[$-3.9\cdot10^{-4}$, $3.2\cdot10^{-4}$]	0.0236	0.0064	0.0023
	10	$>$ 2h	0.227	3.1	38.01	0.354	3.63	35.63
		0	[-0.037, 0.037]	[-0.004, 0.0039]	[$-4.2\cdot10^{-4}$, $3.5\cdot10^{-4}$]	0.017	0.007	0.005
Rand50-AVG[1]	4	$>$ 2h	1.1544	19.77	390.58	5.676	55.021	590.17
		0	0.007	0.0007	0	0.0243	0.0084	0.0024

Table 2: Runtime and errors estimation comparison (run time in seconds) for sequential plans

tion algorithm (which occurred sometimes when only low accuracy was required), when increasing the required accuracy for both algorithms, eventually the approximation algorithm overtook the sampling algorithm.

The graphs in figures 4, 5 and 6 present more vividly the results for the task trees used as execution plans in the DARPA robotics challenge. The horizontal axis represents the run-time of the algorithm (exact, sampling, approximation) and the vertical axis represent the upper bound error. It is easy to see that in all graphs our approximation algorithm for $\varepsilon = 0.01$ and $\varepsilon = 0.001$ provides very small error compared to the exact result and the run-time is lower. In addition, in most graphs (7/8), the sampling results are less accurate and the run-time is higher compared to our approximation algorithm results. The only exception is in the Drive task tree with $M = 10$ and $\varepsilon = 0.1$, our result is less accurate and the run-time is higher than the sampling algorithm for 10^3 samples. However, for 10^4 the run-time of our approximation algorithm is lower than the sampling algorithm and for $\varepsilon = 0.01$ and $\varepsilon = 0.001$ our results are better than the sampling results.

Discussion

There are numerous minor issues related to the work presented above, briefly discussed here. One set of issues is related to trivial improvements to the `Trim` operator, such as the inverse version of the operator used to generate a lower bound for the empirical results. Other candidate improvements are not performing trimming (or even stopping a trimming operation) if the current support size is below $1/\varepsilon$, which may increase accuracy but also the runtime. Another point is that in the combined algorithm, space and time complexity can be reduced by adding some `Trim` operations, especially after processing a parallel node, which is not done in our version. This may reduce accuracy, a trade-off yet to be examined. Another option is, when given a specific threshold, trying for higher accuracy in just the region of the threshold, but how to do that is non-trivial. For *sampling* schemes such methods are known, including adaptive sampling (Bucher 1988; Lipton, Naughton, and Schneider 1990), stratified sampling, and other schemes. It may be possible to apply such schemes to deterministic algorithms as well - an interesting issue for future work.

Incremental execution [2] of our algorithm can also be considered as future work, for instance, when a plan is being generated, allow a planner to backtrack when it discovers the current plan candidate is with high probability to exceed the given deadline or to guide the search process in other ways.

An obvious extension is handling continuous distributions. In practice, our algorithm can handle continuous distributions by pre-running a version of the `Trim` operator on the primitive task distribution. Since one cannot iterate over support values in a continuous distribution, start with the smallest support value (even if it is $-\infty$), and find the value at which the CDF increases by ε. This requires access to the inverse of the CDF, which is available, either exactly or approximately, for many types of distributions. In fact, this is precisely how Gaussian distributions were handled in our empirical evaluation.

Approximation algorithm for the deadline problem is the main focus of this paper. We also presented some results for the expectation problem, mainly showing that this problem is also NP-hard. A natural question is on approximation algorithms for the expectation problem, but the answer here is not so obvious. Sampling algorithms may run into trouble if the target distribution contains major outliers, i.e. values very far from other values but with extremely low probability. Our approximation algorithm can also be used as-is to estimate the CDF and then to approximate the expectation, but we do not expect it to perform well because our current `Trim` operator only limits the amount of probability mass moved at each location to ε, but does not limit the "distance" over which it is moved. The latter may be arbitrarily bad for estimating the expectation. Possibly adding simple binning schemes to the `Trim` operator in addition to limiting the moved probability mass to ε may work, another issue for future research.

In this paper, we focus mostly on the issue of computing the probability $P(t < T)$ of satisfying a deadline T, i.e., that the makespan t of the plan is less than a given value. Related work on computing makespan distributions includes (Hong 2013), which examines sum of Bernoulli distributed random variables. Other work examines both deterministic (Mercier 2007) and Monte-Carlo techniques (Bucher 1988; Lipton, Naughton, and Schneider 1990). Distribution of maximum of random variables was studied in (Devroye 1980), with a focus mostly on continuous distributions.

Complexity of finding the probability that the makespan is under a given threshold in task networks was shown to be NP-hard in (Hagstrom 1988), even when the completion time of each task has a Bernoulli distribution. Nevertheless, our results are orthogonal as the source of the complexity in (Hagstrom 1988) is in the graph structure, whereas in our setting the complexity is due to the size of the support. In fact for linear plans (an NP-hard case in our setting), the probability of meeting the deadline can be computed in low-order polynomial time for Bernoulli distributions, using straightforward dynamic programming. Makespan distributions in series parallel networks in the i.i.d. case was examined in (Gutjahr and Pflug 1992), without considering algorithmic issues. There is also a significant body of work on estimating the makespan of plans and schedules (Herroelen and Leus 2005; Fu, Varakantham, and Lau 2010; Beck and Wilson 2007), within a context of a planner or scheduler. The analysis in these papers is based on averaging or on limit theorems, and does not provide a guaranteed approximation scheme.

Computing the distribution of the makespan in trees is a seemingly trivial problem in probabilistic reasoning (Pearl 1988). Given the task network, it is straightforward to represent the distribution using a Bayes network (BN) that has one node per task, and where the *children* of a node v in the task network are represented by BN nodes that are *parents* of the BN node representing v. This results in a tree-shaped BN, where it is well known that probabilistic rea-

[2] Suggested by an anonymous reviewer

soning can be done in time linear in the number of nodes, e.g. by belief propagation (message passing) (Pearl 1988; Kim and Pearl 1983). The difficulty is in the potentially exponential size of variable domains, which our algorithm, essentially a limited form of approximate belief propagation, avoids by trimming.

Looking at makespan distribution computation as probabilistic reasoning leads to interesting issues for future research, such as how to handle task completion times that have dependencies, represented as a BN. Since reasoning in BNs is NP-hard even for binary-valued variables (Dagum and Luby 1993; Cooper 1990), this is unlikely in general. But for cases where the BN topology is tractable, such as for BNs with bounded tree width (Bodlaender 2006), or directed-path singly connected BNs (Shimony and Domshlak 2003), a deterministic polynomial-time approximation scheme for the makespan distribution may be achievable. The research literature contains numerous *randomized* approximation schemes that handle dependencies (Pearl 1988; Yuan and Druzdzel 2006), especially for the case with *no evidence*. In fact, our original implementation of the sampling scheme in our ROBIL team entry handled dependent durations. It is unclear whether such sampling schemes can be adapted to handle dependencies *and* arbitrary evidence, such as: "the completion time of compound task X in the network is known to be exactly 1 hour from now". Finally, one might consider additional commonly used utility functions, such as a "soft" deadline: the utility is a constant U before the deadline T, decreasing linearly to 0 until $T + G$ for some "grace" duration G, and 0 thereafter.

Acknowledgments. This research was supported by the ROBIL project, by the EU, by the ISF, and by the Lynne and William Frankel Center for Computer Science.

References

Beck, J. C., and Wilson, N. 2007. Proactive algorithms for job shop scheduling with probabilistic durations. *J. Artif. Intell. Res.(JAIR)* 28:183–232.

Bodlaender, H. L. 2006. Treewidth: Characterizations, applications, and computations. In *Proceedings of the 32Nd International Conference on Graph-Theoretic Concepts in Computer Science*, WG'06, 1–14. Berlin, Heidelberg: Springer-Verlag.

Bonfietti, A.; Lombardi, M.; and Milano, M. 2014. Disregarding duration uncertainty in partial order schedules? Yes, we can! In *Integration of AI and OR Techniques in Constraint Programming*. Springer. 210–225.

Bucher, C. G. 1988. Adaptive sampling: an iterative fast Monte Carlo procedure. *Structural Safety* 5(2):119–126.

Cooper, G. F. 1990. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence* 42 (2-3):393–405.

Dagum, P., and Luby, M. 1993. Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial Intelligence* 60 (1):141–153.

Devroye, L. 1980. Generating the maximum of independent identically distributed random variables. *Computers & Mathematics with Applications* 6(3):305–315.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *AAAI*, volume 94, 1123–1128.

Fu, N.; Varakantham, P.; and Lau, H. C. 2010. Towards finding robust execution strategies for RCPSP/max with durational uncertainty. In *ICAPS*, 73–80.

Gabaldon, A. 2002. Programming hierarchical task networks in the situation calculus. In *AIPS02 Workshop on Online Planning and Scheduling*.

Garey, M. R., and Johnson, D. S. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co.

Gutjahr, W., and Pflug, G. C. 1992. Average execution times of series–parallel networks. *Séminaire Lotharingien de Combinatoire* 29:9.

Hagstrom, J. N. 1988. Computational complexity of PERT problems. *Networks* 18(2):139–147.

Herroelen, W., and Leus, R. 2005. Project scheduling under uncertainty: Survey and research potentials. *European journal of operational research* 165(2):289–306.

Hong, Y. 2013. On computing the distribution function for the poisson binomial distribution. *Computational Statistics & Data Analysis* 59:41–51.

Kelly, J. P.; Botea, A.; and Koenig, S. 2008. Offline planning with Hierarchical Task Networks in video games. In *AIIDE*, 60–65.

Kim, J. H., and Pearl, J. 1983. A computation model for causal and diagnostic reasoning in inference systems. In *Proceedings of the 6th International Joint Conference on AI*.

Lipton, R. J.; Naughton, J. F.; and Schneider, D. A. 1990. *Practical selectivity estimation through adaptive sampling*, volume 19. ACM.

Mainprice, J.; Sisbot, E. A.; Jaillet, L.; Cortés, J.; Alami, R.; and Siméon, T. 2011. Planning human-aware motions using a sampling-based costmap planner. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 5012–5017. IEEE.

Mercier, S. 2007. Discrete random bounds for general random variables and applications to reliability. *European journal of operational research* 177(1):378–405.

Nau, D. S.; Smith, S. J.; Erol, K.; et al. 1998. Control strategies in HTN planning: Theory versus practice. In *AAAI/IAAI*, 1127–1133.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *J. Artif. Intell. Res. (JAIR)* 20:379–404.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann.

Shimony, S. E., and Domshlak, C. 2003. Complexity of probabilistic reasoning in directed-path singly connected Bayes networks. *Artificial Intelligence* 151:213–225.

Yuan, C., and Druzdzel, M. J. 2006. Importance sampling algorithms for Bayesian networks: Principles and performance. *Mathematical and Computer Modelling* 43(910):1189 – 1207.

Planning with State Constraints and its Application to Combined Task and Motion Planning

Jonathan Ferrer-Mestres
Universitat Pompeu Fabra
Barcelona, Spain
jonathan.ferrer@upf.edu

Guillem Francès
Universitat Pompeu Fabra
Barcelona, Spain
guillem.frances@upf.edu

Hector Geffner
ICREA & Universitat Pompeu Fabra
Barcelona, Spain
hector.geffner@upf.edu

Abstract

Most of the key computational ideas in planning have been developed for planning languages where action preconditions and goals are conjunctions of propositional atoms. These restrictions make the definition and computation of planning heuristics easier but limit the expressive capabilities of the resulting planners. As a result, standard planners are unable to capture the type of geometrical reasoning that is critical in robotics domains where both robots and objects have geometrical dimensions and collisions are to be avoided. Such problems are addressed in robotics by combining *task planners* that handle the symbolic reasoning part with *motion planners* that check the geometrical feasibility of the plans output by the task planners. This decomposition however may result in a lot of backtracking as the symbolic and geometrical components are not independent. The aim of this work is to provide an alternative integration of task and motion planning where the symbolic and geometrical components are addressed in combination, with neither part taking the back seat. For this, we build on the recent planner `FS0` that is able to handles an expressive variable-free, first-order planning language called Functional STRIPS where constraints, functions and numerical variables are accommodated, and extend it to handle also *state constraints* — namely, formulas that must be true in all states. We then use *functions* for encoding the geometrical dimensions and poses of objects, and *state constraints* to express that no pair of objects, including the robot, can overlap in space. We illustrate the functionality and performance of the planner over a number of examples.

Introduction

Classical AI planners are currently able to solve problems over very large state spaces. In classical planning, the initial state is fully known and actions are assumed to have deterministic effects. The main techniques rely on the use of heuristics that are derived automatically in order to guide a state-space search or on translations into propositional satisfiability (Russell and Norvig 2002; Ghallab, Nau, and Traverso 2004; Geffner and Bonet 2013). However, the restrictions in planning modeling languages that have facilitated these developments, have also limited the scope of the resulting planners. In particular, standard planners do not appear to be suitable for capturing the type of geometrical reasoning that is critical in robotics where both robots and objects have geometrical dimensions and collisions are to be avoided. Such problems are addressed instead through a combination of two types of planners: task planners that handle the high-level, symbolic reasoning part, which correspond to the AI planners, and motion planners (LaValle 2006) that plan the movements in space and handle the geometrical constraints (Cambon, Gravot, and Alami 2004; Wolfe, Marthi, and Russell 2010; Lagriffoul et al. 2012; Lozano-Pérez and Kaelbling 2014). The symbolic and the geometrical components, however, are not independent, and hence, by giving one of these two parts the secondary role of verifying feasibility, approaches based on task and motion decomposition are doomed to produce lots of backtracks. The problem of excessive backtracks is well-known in constraint satisfaction when constraints are used *passively* (Mackworth 1977; Dechter 2003). The computational solution to this problem is the interleaving of search with forms of constraint propagation that make the constraints an *active* part in the search. The situation is similar when searching for a goal in a graph: blind-search methods where the goal plays a passive role cannot scale as well as methods where the goal directs the search by means of a heuristic function.

The aim of this work is to provide an alternative integration of task and motion planning where the symbolic and geometrical components in robot planning are addressed in combination and both parts play an active role in directing the search for plans. For this, we build on the recent planner `FS0` (Francès and Geffner 2015) that handles an expressive variable-free, first-order planning language called Functional STRIPS (Geffner 2000) which naturally accommodates constraints, functions, and numerical variables *both in the specification of the problem and in the computation of the heuristics*. On top of that, we extend `FS0` and Functional STRIPS with the ability to handle *state constraints*, *i.e.* formulas that must be true in all states encountered through the execution of a plan. We use *functions* for encoding the geometrical dimensions of objects and their poses, and *state constraints* to express that no two objects, including the robot, can overlap in space. State constraints are not a standard feature of planning languages, although their convenience has been thoroughly discussed in the literature on reasoning about actions (Lin and Reiter 1994; Son et al. 2005), sometimes under the name of *plan invariants*.[1] In our case, state constraints can be thought of as a

[1] State constraints should not be confused with state invariants: a

convenient way to express preconditions that are implicit for all actions (*i.e.* actions leading to the violation of a state constraint are deemed not applicable), states to be avoided (*e.g.*, states where a formula $p \wedge q$ is true),[2] and dead-end conditions, that is conditions that if ever achieved preclude reaching the goal. Functions are then used in the planner to encode the geometrical dimensions of objects and their poses, and state constraints to express that objects should not overlap in space. The functionality and performance of the planner will be illustrated over some examples.

The rest of the paper is organized as follows. We first review the Functional STRIPS planning language, whose expressivity is key for handling "motion planning" as a particular form of "task planning", and the recent planner `FS0`, which handles a large fragment of Functional STRIPS. We then show how to extend this planner for handling state constraints. Finally, we use the resulting planner for modeling and solving planning problems with state constraints, and in particular, problems that combine motion and task planning. Preliminary experimental results are reported, and simulations of the resulting plans can be seen in videos.[3]

Functional STRIPS

Functional STRIPS (FSTRIPS) is a general modeling language for classical planning based on the quantifier-free fragment of first-order-logic involving *constant*, *function* and *relational or predicate symbols* but no variable symbols. We review it following (Geffner 2000).

Syntax

FSTRIPS assumes that *fluent* symbols, whose denotation may change as a result of the actions, are all *function* symbols. Fluent constant symbols can be seen as arity-0 function symbols, and fluent relational symbols as *boolean* function symbols of the same arity plus equality. For example, typical blocksworld atoms like $on(a, b)$ can be encoded in FSTRIPS as $on(a, b) = true$, by making on a functional symbol, or in this case, more conveniently, as $loc(a) = b$ where loc is a function symbol denoting the block location.

Constant, functional and relational symbols whose denotation does not change are called *fixed* symbols. Among them, there is usually a finite set of object names, and constant, function, and relational symbols such as '3', '+' and '=', with the standard interpretation. Terms, atoms, and formulas are defined from constant, function, and relational symbols in the standard way, except that in order for the representation of states to be finite and compact, the symbols, and hence the terms are typed. A *type* is given by a finite set of fixed constant symbols. The terms $f(t)$ where f is a fluent symbol and t is a tuple of fixed constant symbols are called *state variables*, as the state is actually determined by the value of such "variables".

An action a is described by the type of its arguments and two sets: the *precondition* and the *effects*. The precondition $Pre(a)$ is a formula, and the effects are *updates* of the form $f(t) := w$, where $f(t)$ and w are terms of the same type, f is a fluent symbol, and t is a tuple of terms. The updates express how fluent f changes when the action is taken. Conditional effects $C \rightarrow f(t) := w$, where C is a formula (possibly $C = true$), can be defined in a similar manner.

A FSTRIPS problem is a tuple $P = \langle F, I, O, G \rangle$ where I and G are the initial and goal formulas, O is a set of actions, and F describes the symbols and their types. The formula I along with the external procedures must define a unique initial denotation for each of the symbols in F.

Semantics

States represent logical interpretations over the language of FSTRIPS. The denotation of a symbol or term t in the state s is written as t^s. The denotation r^s of fixed symbols r does not depend on the state and is written r^*. The denotation of standard fixed symbols like '3', '+', '=' is assumed to be given by the underlying programming language, while object names c are assumed to denote themselves *i.e.* $c^* = c$. The denotation of fixed (typed) function and relational symbols can be provided extensionally, by enumeration in the initial situation, or intensionally, by attaching actual functions (external procedures) to them (Dornhege et al. 2009).

Since the only fluent symbols are function symbols, and the types of their arguments are all finite, the (dynamic part of the) state can be represented as the value of a finite set of state variables $f(t)$, where f is a functional fluent and t is a tuple of fixed constant symbols. From the fixed denotation r^* of fixed symbols r, and the changing denotation of fluent symbols f captured by the values $[f(t)]^s$ of the state variables $f(t)$ associated with f, the denotation of arbitrary terms, atoms, and formulas follows in the standard way. The denotation t^s of any term not involving functional fluents, expressed also as t^*, is c^* if t is a constant symbol or, recursively, $g^*(t_1^*)$ if t is the compound term $g(t_1)$ where t_1 is a tuple of terms. Similarly, the denotation t^s of a term $f(t_1)$ where f is a fluent functional symbol is defined recursively as the value $[f(c)]^s$ of the *state variable* $f(c)$ in s where c is the tuple of constant symbols that name the tuple of objects t_1^s; i.e., $c^* = t_1^s$. In the same way, the denotation $[p(t)]^s$ of an atom $p(t)$ is $true/false$ iff the result of applying the boolean function p^* to the tuple of objects t^s yields $true/false$. The truth value B^s of the formulas B made up of such atoms in the state s follows then the usual rules.

An action a is applicable in a state s if $[Pre(a)]^s = true$. The state s_a that results from the action a in s satisfies the equation $f^{s_a}(t^s) = w^s$ for all the updates $f(t) := w$ that the action a triggers in s, and is otherwise equal to s. This means that the update changes the value of the *state variable* $f(c)$ to w^s iff the action triggers an update $f(t) := w$ in the state s for which $c^* = t^s$. For example, if $X = 2$ is true in s, then the update $X := X + 1$ increases the value of X to 3 without affecting other state variables. Similarly, if

state constraint *enforces* a formula to be true in all reachable states, while a state invariant is a formula that can be proven to hold in all reachable states. For example, block A not being on top of two blocks at the same time is a typical state invariant in blocksworld, whereas A not being on top of block B might be a particular state constraint that we want to enforce.

[2] State constraints ϕ used this way model the class of extended temporal goals "never ϕ" (Dal Lago, Pistore, and Traverso 2002).

[3] `www.bitbucket.org/ferrerj/ctmp`

$loc(b) = b'$ is true in s, the update $clear(loc(b)) := true$ in s is equivalent to the update $clear(b') := true$.

A plan for a problem $P = \langle F, I, O, G\rangle$ is a sequence of applicable actions from O that maps the unique initial state where I is true into one of the states where G is true.

Example

A simple planning problem involving a set of integer variables $X_1, \ldots, X_n$ and actions that allow us to increase or decrease the value of any variable by one within the $[0, n]$ interval, can be modeled in Functional STRIPS by treating the variables X_i as 0-arity fluent functional symbols with values ranging in the $[0, n]$ interval. These X_i symbols represent the state variables in the problem. If $I = \{X_1 = 0, \ldots, X_n = 0\}$, and $G = \{X_1 < X_2, \ldots, X_{n-1} < X_n\}$, the problem is about changing the value of the X_i variables from 0 to final values that increase monotonically with i. The precondition-free action $inc(X_i)$ has the effect $X_i := \min(X_i + 1, n)$, whereas $dec(X_i)$ has the effect $X_i := \max(X_i - 1, 0)$.

A Functional STRIPS Planner

The `FS0` planner (Francès and Geffner 2015) deals with a large fragment of the Functional STRIPS language where preconditions and goal formulas must be in CNF and each clause may involve two state variables at most, with the exception of atoms expressing *global constraints* (van Hoeve and Katriel 2006; Rossi, Van Beek, and Walsh 2006). For example, the goal of arranging a set of blocks in blocksworld into a single tower can be expressed succinctly as $\texttt{alldiff}(loc(b_1), \ldots, loc(b_n))$ where `alldiff` represents the standard *all-different* constraint. Like constraint solvers, `FS0` gets a computational benefit from combining the $O(n^2)$ inequality constraints $loc(b_i) \neq loc(b_j)$ into one single `alldiff` global constraint, as this enables more powerful forms of constraint propagation that yield more informed heuristic values. We next focus on the computation of these heuristics in `FS0`, where they are used to guide a standard *greedy best-first search*.

Computation of the Heuristic

As noted by several authors, some of the heuristics that are useful in STRIPS like $h_{\max}$ (Bonet and Geffner 2001) and h_{FF} (Hoffmann and Nebel 2001) can be generalized to more expressive languages by means of the so-called *value-accumulating semantics* (Hoffmann 2003; Gregory et al. 2012; Ivankovic et al. 2014). In this interpretation, each propositional layer P_k of the **Relaxed Planning Graph (RPG)** keeps for each state variable X a set X^k of values that are possible in P_k. Such sets are used to define the *sets y^k of possible values or denotations of arbitrary terms, atoms, and formulas y*, and from them, the sets of possible values X^{k+1} for the next layer P_{k+1}. For layer P_0, $X^0 = \{X^s\}$, s being the state whose heuristic value is sought. From the sets of possible values X^k for the state variables X in layer P_k, the set of possible denotations t^k of any term not involving functional fluents is $t^k = \{t^*\}$, while the set of possible denotations $[f(t)]^k$ for terms $f(t)$ where f is a fluent symbol is defined recursively as the *union* of the sets $[f(c)]^k$ where $f(c)$ is a state variable such that $c^* \in t^k$. In a similar way, the set of possible denotations $[p(t)]^k$ of an atom $p(t)$ in layer P_k includes the value $true$ ($false$) iff $p^*(c^*) = true$ ($false$, respectively) for some tuple $c^* \in t^k$. The set of possible denotation of disjunctions, conjunctions, and negations are defined recursively so that $true$ is in $[A \vee B]^k$, $[A \wedge B]^k$ and $[\neg A]^k$ iff $true$ is in A^k or in B^k, $true$ is in both A^k and B^k, and $false$ is in A^k respectively. Similarly, $false$ is in $[A \vee B]^k$, $[A \wedge B]$ and $[\neg A]^k$ iff $false$ is in both A^k and B^k, $false$ is in A^k or in B^k, and $true$ is in A^k respectively. The set of possible values X^{k+1} for the state variable X in layer P_{k+1} is the union of X^k and the set of possible values x for X that are supported by conditional effects of actions a whose preconditions are possible in P_k, i.e., $true \in [Pre(a)]^k$. A conditional effect $C \rightarrow f(t) := w$ of a supports value x of X in P_k iff $X = f(c)$ for some tuple of constant symbols c such that $c^* \in t^k$ and $x \in w^k$. When computing the heuristics $h_{\max}$ and h_{FF}, the computation stops in the first layer P_k where the goal formula G is true, *i.e.* $true \in G^k$, or where a fixed point has been reached without rendering the goal true, *i.e.* $X^k = X^{k+1}$ for all the state variables. A *relaxed plan* $\pi_{\text{FF}}(s)$ can be obtained then backward from the goal by keeping track of the state variables X and values $x \in X^k$ that make the goal true, the actions a and effects $C \rightarrow f(t) := w$ supporting such values first, and iteratively, the variables and values that make $Pre(a)$ and C true. The heuristic $h_{\text{FF}}(s)$ is given by the number of different actions a in $\pi_{\text{FF}}(s)$, each action a counted as many times as layers in $\pi_{\text{FF}}(s)$ where it is used, in accordance with the treatment of conditional effects in `FF`. The heuristic $h_{\max}(s)$ is given by the index k of the first layer P_k where the goal is true.

Logical Generalization: Constrained RPG. A weakness of RPG heuristics is the assumption that *state variables can take several values at the same time.* This simplification does not follow from the *monotonicity assumption* that underlies the value-accumulating semantics but from the way the sets of *possible* values X^k in layer P_k are used. The fact that these various values are all regarded as possible in layer P_k does not imply that they are *jointly* possible. The way to retain monotonicity in the construction of the planning graph while removing the assumption that a state variable can take several values at the same time is *to map the domains X^k of the state variables into a set V^k of possible interpretations over the language*. Indeed, given that an interpretation s over the language is determined by the values X^s of the state variables (Section 2), this set V^k is nothing but the set of interpretations v that result from selecting *one value* X^v for each state variable X among the set of values X^k that are possible for X in layer P_k. As before, $X^0 = \{X^s\}$ when s is the seed state, and X^{k+1} contains all the values in X^k along with the set of possible values x for X supported by the effects of actions a whose preconditions are possible in P_k. A formula like $Pre(a)$ is *satisfiable* in layer P_k iff there is an interpretation $v \in V^k$ s.t. $[Pre(a)]^v = true$. Moreover, a conditional effect $C \rightarrow f(t) := w$ of a supports the value x of X in P_k iff there is an interpretation $v \in V^k$ where $[Pre(a)]^v$ and C^v are $true$, $x = w^v$, and $X = f(c)$ for $c^* = t^v$. This alternative, **logical interpreta-**

tion of the propositional layers P_i affects the contents and computation of the RPG, which we now call **Constrained RPG (CRPG)**. The construction of the RPG stops at the first layer P_k where the goal formula G is satisfiable, *i.e.* where G^v is $true$ for some $v \in V^k$, or when a fixed point is reached without rendering the goal true. Heuristics analogous to h_{max} and h_{FF} (which we name h^*_{max} and h^*_{FF}) can then be obtained from such a graph in the usual way.

Polynomial Approximation: Constraint Propagation. The heuristics h^*_{max} and h^*_{FF} correctly assign infinite heuristic values to logically inconsistent goals such as $(X < 3 \wedge X > 5)$, which get finite values in the unconstrained heuristics h_{max} and h_{FF} when each goal $X < 3$ and $X > 5$ is reachable separately. The problem with such heuristics is that they are *intractable*. The two heuristics h^c_{max} and h^c_{FF} supported in the `FS0` planner are aimed at approximating these two heuristics in an efficient, *polynomial* manner. For this, it is assumed that action preconditions, conditions, and goals are conjunction of atoms, and that fluent symbols do not appear nested. Under these restrictions, checking whether the goal G is *satisfiable* in a propositional layer P_k boils down to solving a *constraint satisfaction problem* G^k whose *variables* are the state variables X appearing in G, the *domain* $D(X)$ of the variables X is X^k, and the constraints are given by the *atoms* in G. In the approximation, the goal G is deemed *satisfiable* simply if the local consistency methods do not prove that the CSP is unsatisfiable by leaving a variable with an empty domain. The local consistency methods are node and arc consistency (Dechter 2003; Rossi, Van Beek, and Walsh 2006). The h^c_{max} heuristic is defined by the index k of the first layer P_k where the goal is deemed satisfiable in this manner. Arc consistency applies only to binary constraints, *i.e.* to atoms involving two state variables. For atoms involving more variables, local consistency algorithms that depend on the type of (global) constraint are used instead. Currently, `FS0` supports just two types of global constraints, `alldiff` and `sum` (Rossi, Van Beek, and Walsh 2006), but it offers the possibility of defining new global constraints by providing their associated local consistency algorithms.

As an illustration of these heuristics, the problem above with variables X_i that must be increased or decreased from 0 until achieving the inequalities $X_i < X_{i+1}, i = 1, \ldots, n-1$, yields an h_{max} value of 1 when the value-accumulating semantics is used, or when the problem is compiled into STRIPS. On the other hand, the constrained heuristic h^*_{max} and its polynomial approximation h^c_{max} have the optimal value n. This is because while each of the goal atoms $X_i < X_{i+1}$ is reachable in one step in the RPG, their *conjunction* is satisfiable in both the constrained RPG and its polynomial approximation only after n steps.

State Constraints

By accommodating numerical variables, constraints, and functions, it is possible to express in Functional STRIPS problems that involve geometrical reasoning. For example, one can easily deal with a problem involving a square object o with side length d by using a representation in which the function $config(o)$ denotes its 2D location $\langle x, y\rangle$ which can be changed through translation actions. By a suitable discretization of the set of possible configurations, one can then express the problem of manipulating the object o from a given initial configuration into a final configuration where, for example, a certain position (x_c, y_c) must be covered by the object. For this, the goal G can be expressed as the formula $(x - d \leq x_c \leq x + d) \wedge (y - d \leq y_c \leq y + d)$ that reduces to four atoms involving two state variables x and y. Actually, the current version of `FS0` handles the negation of such goals as well, which also involves two variables, and expresses the problem where the object must be placed in a location where the target point is not covered. Thus, it is simple to capture in `FS0` problems where an object must be moved in order to comply with some *geometrical goal constraint*. What is less simple to do in Functional STRIPS is to enforce such constraints *throughout the execution* of the plans which is critical in robotics. For doing this, however, we just need to treat such constraints, not as goal constraints, but as *state constraints* (Lin and Reiter 1994; Son et al. 2005), i.e., constraints applying to all states not just goal states. The changes required in the language and in the derivation of heuristics for accommodating state constraints are minor, but the expressivity gained is significant.

Syntax and Semantics

A FSTRIPS planning problem with *state constraints* is a tuple $P = \langle F, I, O, G, C\rangle$ where the new component C stands for a set of formulas expressing the constraints. The syntax for these formulas is the same as for those encoding the goal G but their semantics is different. State constraints are used for encoding *implicit preconditions.* Namely, an action a is deemed applicable in a state s when *both* $[Pre(a)]^s = true$ *and* the state s_a that results from applying a to s is such that $c^s = true$ for every state constraint $c \in C$. In other words, an action a is *non-executable* in a state s where its precondition $Pre(a)$ holds, if its execution leads to a state s_a that violates some state constraint. In addition, the unique initial state must satisfy all the state constraints as well. As a result, if $c \in C$ is a state constraint and $s_0, \ldots, s_n$ is the sequence of states generated by a plan that solves P, then c will be true in all the states $s_i, i = 0, \ldots, n$.

Heuristics

The introduction of state constraints affects the definition of the heuristics h^*_{max} and h^*_{FF} obtained from the constrained RPG, and the polynomial approximations h^c_{max} and h^c_{FF} supported in the `FS0` planner. The changes, however, are minor. First, recall that a layer P_k encodes sets X^k of possible values for each state variable X, which in turn define sets V^k of possible interpretations over the language. An action a is deemed applicable in layer P_k if one such interpretation satisfies the formula $Pre(a)$; similarly, the goal G is taken to be true for the purpose of computing a constrained relaxed plan, if one such interpretation satisfies G. In the presence of state constraints, this remains the same except that interpretations in V^k that do not satisfy a state constraint are pruned first. In the polynomial approximation that leads to the heuristics

h^c_{max} and h^C_{FF}, the state constraints are not used to prune interpretations directly but just the domains X^k of the state variables X in layer k through constraint propagation.

Examples

We illustrate next the usefulness of state constraints both in terms of modeling and computation with a couple of examples, reporting their running times as well.

The **Missionaries and Cannibals** (M&C) problem has received wide attention since the early days of AI (Amarel 1968) as a toy problem that is nevertheless representative of a wider class of transportation-under-constraints problems. In its standard version, the problem places three missionaries and three cannibals on the left bank of a river which they all want to cross. A single boat is available that can hold only two people at a time, regardless of whether they are missionaries or cannibals. Apparently, the missionaries do not want to be outnumbered by the cannibals, be it on either bank of the river or inside of the boat, for fear of the cannibals exercising their defining inclination.[4] The goal is to find an appropriate schedule of river crossings that transports everyone to the right bank of the river in a safe manner.

We model a generalization of the problem for n missionaries and n cannibals ($n \geq 3$) on a complete graph. There are fixed symbols $l_1, \ldots, l_m$ representing the locations, and state variables $nc(l)$ and $nm(l)$ representing the number of cannibals and missionaries at each location l. In addition, the 0-ary functional symbol X represents the current location of the boat. The actions $move(c, m, l)$, with $0 \leq c, m \leq 2$, $c+m \in \{1, 2\}$, move c cannibals and m missionaries in one boat trip from the current location to l. Their preconditions are $c \leq nc(X)$ and $m \leq nm(X)$, and their effects are $X := l$, $nc(l) := nc(l)+c$, $nc(X) := nc(X)-c$, and analogously for the missionaries. Finally, the restriction on cannibals not outnumbering missionaries in a location l is modeled by the binary state constraint $nm(l) \geq nc(l) \vee nm(l) = 0$. The "inside of the boat" restriction is encoded as part of the $move$ action.

As a second example, consider a **simple navigation with geometrical obstacles** problem in which an $n \times m$ grid contains a robot that has to reach a goal cell while avoiding obstacles. For simplicity, we assume a point robot with no geometry, and obstacles o with rectangular shape which can be represented by a couple of coordinate points (x_o, y_o) and (x'_o, y'_o), with $x_o < x'_o$ and $y_o < y'_o$ (obstacles having other shapes can be thought of as a combination of smaller rectangles). The location of the robot is represented by two 0-arity fluent functional symbols x and y with values in $[1, n]$ and $[1, m]$. The actions $move(dx, dy)$ with $dx, dy \in [-1, 1]$ move the agent to adjacent locations, including diagonals, with effects $x := max(0, min(x + dx, n))$ and $y := max(0, min(y + dy, m))$. Avoidance of obstacles o in any plan can then be represented succinctly through the state constraint $\neg(x_o \leq x \leq x'_o \wedge y_o \leq y \leq y'_o)$, resulting in a problem with as many state constraints as obstacles, and where each of the state constraints involves the same two state variables x and y.

[4] A historically more accurate version of the problem has it that it is the cannibals that do not want to be outnumbered by the missionaries for fear of being converted, but we restrict our discussion to the first version for the sake of tradition.

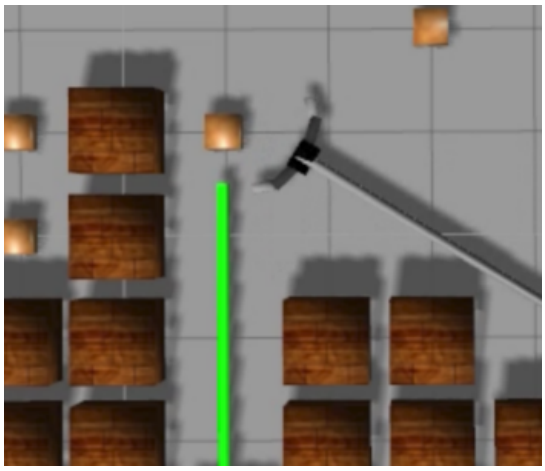


Figure 1: Arm rotating to grasp an object

We have actually tested the planner on a number of randomly generated instances of increasing size for each of these two domains.[5] In the case of the M&C domain, the planner scales up pretty well, handling problem sizes of 20-node location graphs and 12 missionaries plus 12 cannibals with relative ease: an instance with a 10-node graph and $9+9$ missionaries and cannibals is solved in 80 sec. by finding a plan of length 49 after expanding 637 nodes. An instance with 20 nodes and $12 + 12$ individuals takes 379 sec. and 183 node expansions to find a plan of length 45. Similarly, the planner handles navigation problems with a linear number of geometrical obstacles in less than 0.1 sec. for 10×10 grids and less than 15 sec. for 50×50 grids.

Task and Motion Planning Combined

The last example illustrates how problems involving geometrical constraints can be modeled in Functional STRIPS with state constraints, and solved by the `FS0` planner. We now consider a more general type of problem involving a robot that can translate, rotate (Figure 1), and pick and place objects (Figure 2). Robots and objects can have arbitrary 2D geometries, and collisions are to be avoided. In one task, which we name **moving with geometrical obstacles**, the robot must reach a goal configuration by navigating and moving the objects that obstruct the path. In the other task, which we name **tidying up**, the robot must place the objects in some goal configuration. In both cases, the 2D space of the environment is discretized according to a parameter r into a regular grid of size $r \times r$. Robots and objects have a configuration that captures triplets $\langle x, y, \theta \rangle$, where x and y capture the center-of-mass position of the object or robot within the discrete grid, and θ its orientation, with angles being discretized into 45 degrees.

[5] Problem encodings for which empirical results are discussed are available at `www.bitbucket.org/ferrerj/ctmp`.

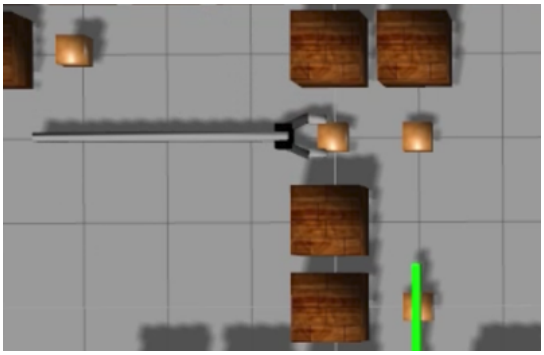


Figure 2: Gripper grasping an object

The configuration $config(o)$ of an object or robot o is represented with the functional fluent $config$. Translations and rotations of the robot change the robot configuration, and, if the robot is holding and object, they change the configuration of the object too. These changes are all expressed by means of externally defined procedures. In addition, the $pickup(o)$ action has precondition $graspable(config(r), config(o))$ where r is the robot, o the object to be picked up, and $graspable$ is a fixed predicate symbol whose denotation is externally defined too.

The *avoidance of collisions between movable and static objects* is captured by the fixed external procedures that perform the configuration updates. If, say, a translation of the robot would make it collide with a static object, the procedure updates the configuration to the particular invalid value $\perp$. The *avoidance of collisions among movable objects*, on the other hand, is handled in a pairwise fashion by the binary state constraints $non\text{-}overlap_{i,j}(config(o_i), config(o_j))$, where the fixed predicate symbol $non\text{-}overlap_{i,j}$ is defined by an external procedure that is a function of the geometries and configurations of the objects i and j. The predicate is true when there is no grid cell occupied by both objects given their fixed geometries and current configurations. For the sake of performance, all reasoning involving object configurations is precompiled into extensional form, meaning that all the fixed external functions are extensionally stored by means of tables that are indexed by the configuration identifiers.

Illustrations showing the initial, goal, and intermediate configurations resulting from plans computed by `FS0` on the "moving with geometrical obstacles" problem are shown in Figure 3. Movable objects are depicted by little squares, static objects by brown boxes, the robot is stick-shaped with a gripper and its final configuration is shown in green.

FSTRIPS Model Details

We provide a few additional details on the model of the problems in FSTRIPS. Besides the $config$ symbol, two auxiliary fluent functional symbols $handempty$ and $held$ are used, the first boolean, the second denoting the object being held. The functions used to capture the changes in the configurations are $translated$, $rotated$ and $o\text{-}rotated$ (standing for the rotation of the object being held). The first function is used in the effects of the translation action, the second in the rotation action, and the third in the rotation action with an object. More precisely, $translated(o, c, d)$ is the configuration that results from applying an *atomic translation* to the robot or object o when in configuration c, where d is a direction; i.e., one of the the fixed symbols N, NE, E, SE, S, SW, W, NW. In turn, $rotated(o, c, d)$ is the result of applying an *atomic rotation* with d being one of the fixed symbols cw and ccw (*clockwise*, *counterclockwise*). Finally, $o\text{-}rotated(o, c, d)$ denotes the resulting configuration of the obstacle o held by the robot when the robot performs a rotation in direction d. By *atomic* translations and rotations we refer to step-wise operations that translate or rotate the object to the next cell or angle respectively, according to the chosen discretization. Having the objects move only one step at a time is necessary to ensure that no collisions happen along trajectories, while keeping the total number of (grounded) actions small and independent of the total number of configurations.

The problem model requires 6 action types `translate`, `rotate` (when the robot is holding no object), `pick-up`, `place`, `translate-with-object`, and `rotate-with-object` (when the robot is holding an object). We omit the full specification and limit our description to the action `rotate-with-object`(o, d), where $d \in \{cw, ccw\}$, and which has a single precondition $held = o$ and two effects: $conf(r) := rotated(r, conf(r), d)$ and $conf(o) := o\text{-}rotated(o, conf(o), d)$. All action preconditions are CNF formulas with clauses involving a single state variable, with the sole exception of the precondition of the `pick-up` action which contains a clause making use of the $graspable$ predicate that involves two state variables.

Experimental Results

We report next the empirical results of the `FS0` planner when run on a number of problems from the MOVING-GEOM-OBSTACLES and TIDYING-UP domains.[6] The results are from running the planner on an AMD Opteron 6300 machine with a 2.4Ghz clock, with a time bound of 30 minutes and a memory bound of 8GB.

Table 1 shows the per-instance results of the planner, focusing on plan length (*i.e.* number of actions in the plan), total number of expanded nodes along the search, and total runtime of the plan search. In both domains, the complexity of the problem clearly grows with both the discretization resolution and the number of movable objects as expected. In general, the higher the value of any of these two parameters, the longer the plans, because steps are smaller and more objects may have to be moved.

MOVING-GEOM-OBSTACLES problems with 10×10 resolution grids are solved by `FS0` with ease, even with 5 obstacles. The number of node expansions during the greedy best-first search grows with the number of objects but is low, suggesting that the heuristic is informative. Grid resolutions of 30×30 and 50×50 pose a more significant challenge, but nevertheless the planner solves the instances with $1 - 3$ objects with a low number of node expansions.

[6] Simulations of the obtained plans can be seen at `www.bitbucket.org/ferrerj/ctmp`.

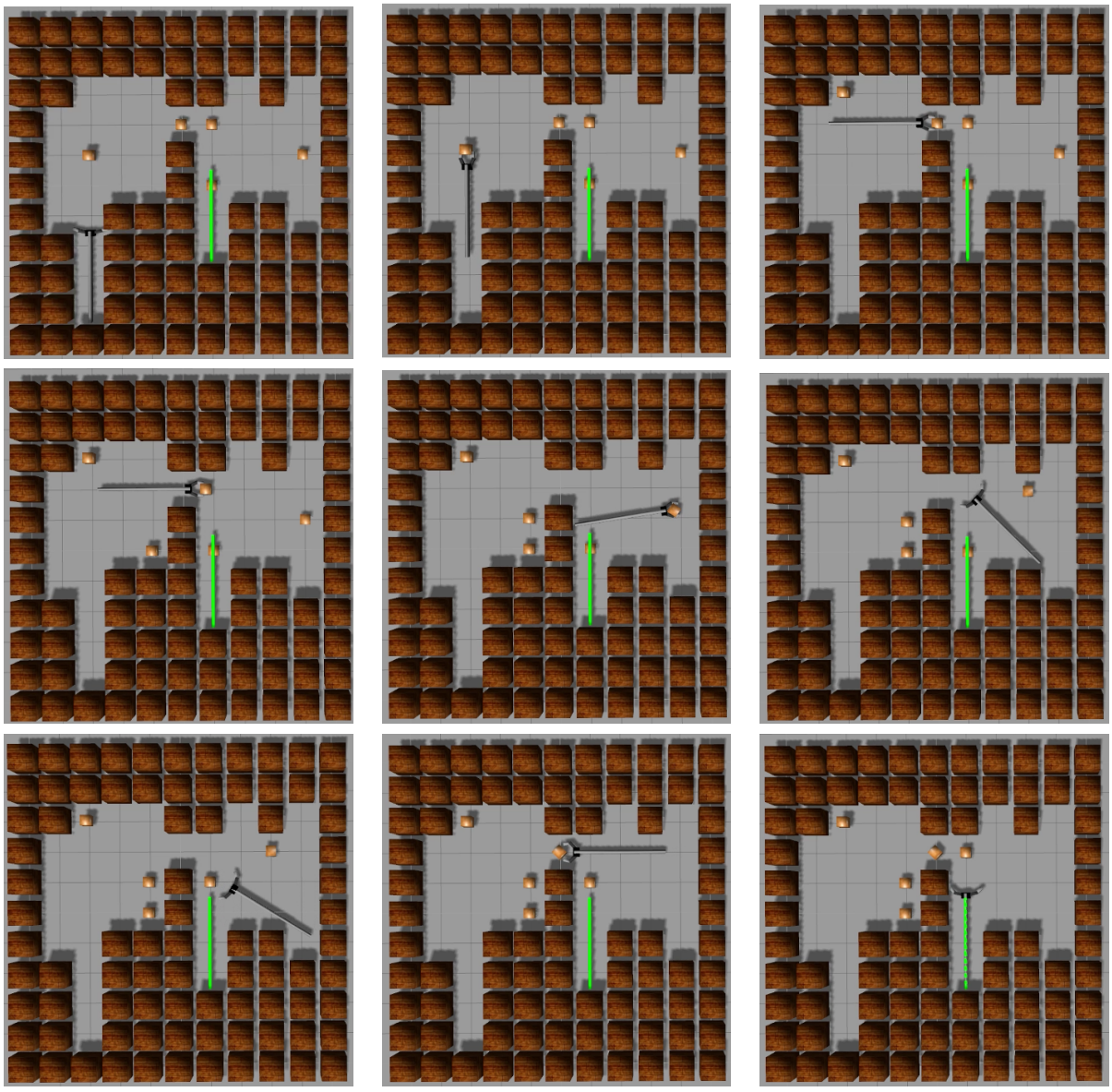


Figure 3: The MOVING-GEOM-OBSTACLES domain in which a robot has to reach a target position but a number of obstacles need to be picked up and moved around so that they stop blocking the robot's path to the goal. Snapshots from top left to bottom right show several steps of the solution, in which the robot reaches the goal configuration after clearing the path.

Problem			FSTRIPS model				Solution			
Type	Res.	Obj.	vars	configs	actions	constr.	length	#Exp	Time (s.)	#Exp per second
MGO	10×10	1	4	212	22	3	18	19	0.16	118.75
MGO	10×10	2	6	212	34	6	19	21	0.45	46.67
MGO	10×10	3	8	212	46	10	23	33	1.13	29.20
MGO	10×10	4	10	212	58	15	41	169	8.10	20.86
MGO	10×10	5	12	212	70	21	59	397	10.56	37.59
MGO	30×30	1	4	894	22	3	44	83	8.35	9.94
MGO	30×30	2	6	894	34	6	49	55	16.39	3.36
MGO	30×30	3	8	894	46	10	55	119	55.26	2.15
MGO	30×30	4	10	894	58	15	-	1475	>1800	0.82
MGO	30×30	5	12	894	70	21	-	2139	>1800	1.19
MGO	50×50	1	4	2120	22	3	74	218	110.43	1.97
MGO	50×50	2	6	2120	34	6	78	80	72.14	1.11
MGO	50×50	3	8	2120	46	10	83	222	548.75	0.40
MGO	50×50	4	10	2120	58	15	-	669	>1800	0.37
MGO	50×50	5	12	2120	70	21	-	1458	>1800	0.81
TU	10×10	2	6	226	34	6	12	14	0.06	233.33
TU	10×10	3	8	227	46	10	35	63	0.43	146.51
TU	30×30	2	6	908	34	6	54	143	13.24	10.80
TU	30×30	3	8	909	46	10	112	6564	164.50	39.90
TU	50×50	2	6	2134	34	6	47	154	5.51	27.95
TU	50×50	3	8	2135	46	10	121	17449	1675.72	10.41

Table 1: Performance of FS0 on MOVING-GEOM-OBSTACLES (MGO) and TIDYING-UP (TU) instances, described in the text. Instances differ on the resolution of the environment discretization (col. 2) and on the number of movable objects (col. 3). Columns 4 to 7 report information about the characteristics of the FSTRIPS encoding, namely the number of state variables, the number of object configurations, the number of (grounded) actions, and the number of state constraints. The four rightmost columns respectively report (1) the length of the plan (a dash denotes that no plan was found), (2) the number of nodes expanded along the search, (3) the total runtime until the plan is found, and (4) the rate at which nodes are expanded. Simulations of the execution of the computed plans can be visualized at www.bitbucket.org/ferrerj/ctmp.

In turn, TIDYING-UP problems are also solved by expanding few nodes in the simpler cases, although the heuristic is less informative for the higher resolutions and 3 objects.

In general, the rate of nodes expanded per second decreases sharply with the resolution and number of objects. This is indicative of the cost of computing the heuristic, which is affected by the number of state variables associated with the objects and by the total number of configurations, which in turn depends on the chosen resolution. Indeed, the number of object configurations in a grid discretized with resolution $r \times r$ and k possible orientations is in the order of $k \cdot r^2$, although many of these configurations will not be feasible, as they overlap with an static object in the map. The empirical results show the feasibility of the approach, but work is still needed to improve scalability further. In particular, one possibility to address the blowup that our approach incurs when increasing the resolution or scaling up to a 3D representation would be the use of an encoding along the lines of the "navigation with geometrical obstacles" example presented above, where configurations are modeled in terms of three distinct state variables x, y and θ instead of one. Particular attention should be paid to how this type of change affects the quality of the resulting heuristics. Another issue that deserves further consideration is the impact of the precompilation of all external procedures into extensional form. This offers an advantage on small-scale domains, but it is not clear how useful it is on larger domains.

Related Work

Combined task and motion planning for grasping and manipulation is an open research problem at the intersection of planning and robotics. The aSyMov planner (Cambon, Gravot, and Alami 2004; Gravot, Cambon, and Alami 2005), for instance, already aimed at integrating both symbolic and geometric reasoning at each step of the planning process. External procedures have also been used to avert the difficulty of performing geometric reasoning within a logically-oriented planning language (Dornhege et al. 2010), as well as to predict the effects of actions involving complex physics (Kunze and Beetz 2015).

Hierarchical approaches to the problem have also been explored in (Kaelbling and Lozano-Pérez 2011), where a hierarchical regression-based schema is developed that combines task and motion planning, and in (Wolfe, Marthi, and

Russell 2010), where Hierarchical Task Networks are used to tackle robotic manipulation problems by modeling the bottom actions of the hierarchy with motion planning. Manipulation planning problems are also tackled through symbolic planning in (Nebel, Dornhege, and Hertle 2013).

An alternative approach is that presented in (Srivastava et al. 2014), where off-the-shelf classical and motion planners are integrated through the use of a planner-independent interface layer. The errors in the motion planning goals have to be identified and fed back to the symbolic planner in the form of logic predicates, the main challenge being the proper identification of the offending atoms that prevent the enactment of specific high-level actions. (Garrett, Lozano-Pérez, and Kaelbling 2014) computes an heuristic that takes the geometrical information into account together with the symbolic information, exploiting a conditional reachability graph as a form of a probabilistic roadmap conditioned to the object configurations. (Lagriffoul et al. 2012) integrates task and motion planning proposing a technique to reduce the geometric configuration space and thus the number of calls to the motion planner. A key difference between all these approaches and ours, however, is *the formulation of motion planning as an additional component of the task planner.*

Summary

We have proposed an alternative integration of task and motion planning where the symbolic and geometrical components are addressed in combination, with neither part taking the back seat. For this, we have built on an expressive planning language, Functional STRIPS, that supports constraints, functions, and numerical variables, and on the planner `FS0`, which supports a large fragment of this language in the specification of problems and is crucially able to exploit its expressivity in the computation of heuristics. We have extended this language and computational model with state constraints: logical formulas that must hold true in every state of a plan. In order to address motion and task planning problems, we use *functions* for encoding the geometrical dimensions of objects and their poses, and *state constraints* to express that no two objects, including the robot, can overlap in space. The experiments reported are preliminary but illustrate the feasibility of the approach. There is a lot of room for improving performance and for exploring the possibilities that are afforded by this integration of motion planning into task planning. In particular, scaling up well in the presence of large grids and many objects remains a challenge. In principle, however, there is no need for the grids to be regular: it would be more natural to use higher resolutions around the current robot configuration and lower resolutions elsewhere. Alternatively, maps obtained from random configuration sampling, as in probabilistic roadmaps, could be used instead. The strength of the integration proposed is that it is very general and independent of these choices.

References

Amarel, S. 1968. On representations of problems of reasoning about actions. *Machine intelligence* 3(3):131–171.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Cambon, S.; Gravot, F.; and Alami, R. 2004. aSyMov: Towards more realistic robot plans. In *Proc. of ICAPS*.

Dal Lago, U.; Pistore, M.; and Traverso, P. 2002. Planning with a language for extended goals. In *Proc. AAAI*, 447–454.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009. Semantic attachments for domain-independent planning systems. In *Proc. of ICAPS*, 114–121.

Dornhege, C.; Eyerich, P.; Keller, T.; Brenner, M.; and Nebel, B. 2010. Integrating task and motion planning using semantic attachments. In *Bridging the Gap Between Task and Motion Planning*.

Francès, G., and Geffner, H. 2015. Modeling and computation in planning: Better heuristics from more expressive languages. In *Proc. of ICAPS*, 70–78. AAAI Press.

Garrett, C. R.; Lozano-Pérez, T.; and Kaelbling, L. P. 2014. FFRob: An efficient heuristic for task and motion planning. In *Proc. Int. Workshop on the Algorithmic Foundations of Robotics (WAFR)*.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool.

Geffner, H. 2000. Functional STRIPS: A more flexible language for planning and problem solving. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer. 187–205.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: theory and practice*. Morgan Kaufmann.

Gravot, F.; Cambon, S.; and Alami, R. 2005. aSyMov: a planner that deals with intricate symbolic and geometric problems. In *International Symposium on Robotics Research*, 100–110. Springer.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning modulo theories: Extending the planning paradigm. In *Proc. of ICAPS*.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J. 2003. The metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research* 20:291–341.

Ivankovic, F.; Haslum, P.; Thiébaux, S.; Shivashankar, V.; and Nau, D. S. 2014. Optimal planning with global numerical state constraints. In *Proc. of ICAPS*.

Kaelbling, L. P., and Lozano-Pérez, T. 2011. Hierarchical task and motion planning in the now. In *International Conference on Robotics and Automation (ICRA)*, 1470–1477. IEEE.

Kunze, L., and Beetz, M. 2015. Envisioning the qualitative effects of robot manipulation actions using simulation-based projections. *Artificial Intelligence*.

Lagriffoul, F.; Dimitrov, D.; Saffiotti, A.; and Karlsson, L. 2012. Constraint propagation on interval bounds for dealing with geometric backtracking. In *International Conference on Intelligent Robots and Systems (IROS)*, 957–964. IEEE.

LaValle, S. M. 2006. *Planning algorithms*. Cambridge.

Lin, F., and Reiter, R. 1994. State constraints revisited. *Journal of logic and computation* 4(5):655–677.

Lozano-Pérez, T., and Kaelbling, L. P. 2014. A constraint-based method for solving sequential manipulation planning problems. In *International Conference on Intelligent Robots and Systems (IROS)*, 3684–3691. IEEE.

Mackworth, A. K. 1977. Consistency in networks of relations. *Artificial intelligence* 8(1):99–118.

Nebel, B.; Dornhege, C.; and Hertle, A. 2013. How much does a household robot need to know in order to tidy up? In *Proceedings of the AAAI Workshop on Intelligent Robotic Systems*.

Rossi, F.; Van Beek, P.; and Walsh, T. 2006. *Handbook of constraint programming*. Elsevier.

Russell, S., and Norvig, P. 2002. *Artificial Intelligence: A Modern Approach*. Prentice Hall. 2nd Edition.

Son, T. C.; Tu, P. H.; Gelfond, M.; and Morales, A. 2005. Conformant planning for domains with constraints: A new approach. In *Proc. AAAI-05*, 1211–1216.

Srivastava, S.; Fang, E.; Riano, L.; Chitnis, R.; Russell, S.; and Abbeel, P. 2014. Combined task and motion planning through an extensible planner-independent interface layer. In *International Conference on Robotics and Automation (ICRA)*, 639–646. IEEE.

van Hoeve, W.-J., and Katriel, I. 2006. Global constraints. *Handbook of constraint programming* 169–208.

Wolfe, J.; Marthi, B.; and Russell, S. J. 2010. Combined task and motion planning for mobile manipulation. In *Proc. of ICAPS*, 254–258.

Continuous Arvand: Motion Planning with Monte Carlo Random Walks

Weifeng Chen and Martin Müller
Department of Computing Science
University of Alberta
{weifeng3,mmueller}@ualberta.ca

Abstract

Sampling-based approaches such as Probabilistic Roadmaps and Rapidly-exploring Random Trees are very popular in motion planning. Monte Carlo Random Walks (MRW) are a quite different sampling method. They were implemented in the Arvand family of planners, which have been successful in classical planning with its discrete state spaces and actions. The work described here develops an MRW approach for domains with continuous state and action spaces, as encountered in motion planning. Several new algorithms based on MRW are introduced, implemented in the Continuous Arvand system, and compared with existing motion planning approaches in the Open Motion Planning Library (OMPL).

1 Introduction

Motion planning refers to breaking down a movement task into discrete motions that satisfy movement constraints. For example, to pick up an object in an environment, the arm of the robot must move to the target location by using its existing actuators, and without colliding with other objects. Motion planning has many applications including robot navigation, manipulation, animating digital characters, automotive assembly and video game design (LaValle 2006).

Among the many approaches to the motion planning problem, sampling based methods have been very popular. A large number of these methods sample randomly from the state space, which is usually called configuration space, or short C-space, in motion planning. The Probabilistic Roadmaps (PRM) (Kavraki et al. 1996) algorithm constructs a roadmap, which connects random milestones, in order to approximate the connectivity of the configuration space. RRT (LaValle and Kuffner 2001) gradually builds a tree that expands effectively in C-space. EST (Hsu, Latombe, and Motwani 1997) attempts to detect the less explored area of the space through the use of a grid imposed on a projection of C-space.

In contrast to sampling from C-space directly, KPIECE (Şucan and Kavraki 2010) is a tree-based planner that explores a continuous space from the given starting point. KPIECE uses a multi-level grid-based discretization for guidance. Given a projection of state space, KPIECE samples cell chains in each iteration when building the exploring tree. The goal of KPIECE is to estimate the coverage of the state space by looking at the coverage of the different cells, and reduce the time used for forward propagation.

Two main criteria for motion planning are feasibility and optimality of plans. The motion planners mentioned above all return the first feasible plan they find. In contrast, planners such as RRT* keep improving their best plan over time, and some are proven to be asymptotically optimal (Karaman and Frazzoli 2011).

Monte Carlo Random Walks (MRW) are the basis for a successful family of algorithms for classical deterministic planning with discrete states and actions (Nakhost and Müller 2009; Nakhost, Hoffmann, and Müller 2012; Nakhost and Müller 2013). The method uses random exploration of the local neighbourhood of a search state. Different MRW variants have been implemented in the Arvand planning systems. The current work applies MRW to continuous planning, using a local sampling forward search framework which, like KPIECE, does not require sampling globally from C-space.

Component	Classical planning	Motion planning
State space	discrete	continuous
Goal checker	deterministic	approximate
Action execution	instant	gradual
Random walk	sample action $\rightarrow$ new state	sample state $\rightarrow$ new motion
Heuristic	Instance-specific, e.g. Fast Forward	C-space-specific, e.g. geometric distance

Table 1: Main differences between using MRW in classical and motion planning.

The high-level view of MRW for continuous planning is similar to classical planning: Random walks are used to explore the neighbourhood of a state and to escape from local minima. A heuristic function which estimates goal distance is used to evaluate sampled states. The main differences between MRW for classical and continuous planning lie in the mechanisms for action selection and action execution within the random walks. In classical planning, for each state s in a random walk, the successor state s' is found by randomly sampling and executing a legal action in s. In contrast, in continuous planning random actions are not generated di-

rectly. Instead, a nearby successor state s' is sampled locally from the state space, and the motion planner is invoked to try to generate a valid motion from s to s'. In classical planning, actions take effect instantly. The solution to a planning problem is simply an action sequence that achieves a goal condition. In continuous planning, each motion action takes time to complete. A solution is a sequence of valid, collision-free motions that get "close enough" to a goal. Table 1 summarizes some main differences of applying MRW to classical and motion planning.

The remainder of this paper is organized as follows: Section 2 describes the main ideas of MRW planning and their application to continuous planning, including different restart strategies and variants for bidirectional search and continuous plan improvement. Section 3 describes the implementation of MRW planning algorithms in the Continuous Arvand system. Section 4 evaluates the performance of the new planners on planning benchmarks from OMPL (Şucan, Moll, and Kavraki 2012). Section 5 is dedicated to concluding remarks and some potential directions for future work.

2 Applying MRW to Continuous Planning

Arvand (Nakhost and Müller 2009; Nakhost, Hoffmann, and Müller 2012; Nakhost and Müller 2013) is a successful family of stochastic planners in classical planning. These planners use Monte Carlo random walks to explore the neighbourhood of a search state. In this work, a similar approach is developed for continuous planning, and implemented in the Continuous Arvand system.

Monte Carlo Random Walk Planning

A MRW algorithm uses the following key ingredients:

- A *heuristic function* h to evaluate the goal distance for the endpoints of random walks. Strong heuristics lead to better performance.
- A *global restart strategy* is used to escape from local minima and plateaus.
- A *local restart strategy* is used for exploration.

In MRW, given a current state s, a number of random walks sample a relatively large set of states S in the neighbourhood of s: the endpoints of each walk. All states in S are evaluated by the heuristic function h. Finally, a new state $s \in S$ with minimum h-value is selected as the next current state, concluding one *search step*, and the process repeats from there. The length of each random walk is decided by the local restart strategy, and could be fixed or variable. Different choices will be discussed below. If the best observed h-value does not improve after a number of search steps, as controlled by the global restart strategy, the search will restart. A good global restart strategy can quickly escape from local minima, and recover from areas of the state space where the heuristic evaluation is poor. The MRW approach does not rely on any assumptions about local properties of the search space or heuristic function. It locally explores the state space before it commits to an action sequence that leads to the best explored state.

MRW Algorithm

Algorithm 1, slightly adapted from (Nakhost and Müller 2009), shows an outline of MRW planning. This high-level outline is nearly identical for classical and for continuous planning. The only change is that a goal condition G is replaced by a goal region G.

The algorithm uses a forward-chaining search in the state space of the problem to find a solution. The chain of states leads from initial state s_0 to goal state s_n. Each transition $s_j \rightarrow s_{j+1}$ is generated by MRW exploring the neighbourhood of s_j. If the best h-value does not improve after a given number of search episodes, MRW simply restarts from s_0.

Algorithm 1 Monte Carlo Random Walk Planning

Input Initial State s_0, goal region G
Output A solution plan

```
s ← s_0
h_min ← h(s_0)
counter ← 0
while s does not satisfy G do
  if counter > MAX_EPISODES then
    s ← s_0 {restart from initial state}
    counter ← 0
  end if
  s ← randomWalk(s, G)
  if h(s) < h_min then
    h_min ← h(s)
    counter ← 0
  else
    counter ← counter + 1
  end if
end while
return the plan reaching the state s
```

Pure Random Walks

The main motivation for MRW planning is to better explore the local neighbourhood, compared to the greedy search algorithms which have been the standard in classical planning. The simplest MRW approach uses a fixed number of pure random walks to sample the neighborhood of a state s. Algorithm 2 shows a pure random walk method similar to the one in (Nakhost and Müller 2009), but adapted to the case of continuous planning. In classical planning, a random legal action is sampled given a current state s' in a random walk, and then applied to reach the next state s''. For continuous planning, instead of an action, the next state is sampled from a region of the state space near s'. Before s'' can succeed s' as the current state, a check is performed to make sure there is a valid motion from s' to s''. A random walk stops either when a goal state is directly reachable, or when the number of consecutive motions reaches a bound *LENGTH_WALK*. The end state of each random walk is evaluated by the heuristic h. The algorithm terminates when either a goal state is reached, or *NUM_WALK* walks have been completed. The function returns the state s_{min} with minimum h-value among all reached endpoints, and the state

sequence leading to it. If no improvement was found, the algorithm simply returns s.

The chosen limits on the length and number of random walks have a huge impact on the performance of this algorithm. Good choices depend on the planning problem. While they are constant in the basic algorithm shown here, the next subsection discusses different adaptive global and local restart strategies, which are used by Arvand and can be applied in continuous planning as well.

Algorithm 2 Pure Random Walks.

Input current state s, goal region G and state space S
Output s_{min}

```
1: h_min ← ∞
2: s_min ← NULL
3: g ← sampleFromGoalRegion(G)
4: for i ← 1 to NUM_WALK do
5:    s' ← s
6:    for j ← 1 to LENGTH_WALK do
7:       if validMotion(s', g) then
8:          return g
9:       end if
10:      repeat
11:         s'' ← uniformlySampleFromNear(s', S)
12:      until validMotion(s', s'')
13:      s' ← s''
14:   end for
15:   if h(s') < h_min then
16:      s_min ← s'
17:      h_min ← h(s')
18:   end if
19: end for
20: if s_min = NULL then
21:    return s
22: else
23:    s_min
24: end if
```

Global and Local Restart Strategy

MRW parameters such as the number and length of random walks, and the maximum number of search episodes, are tedious to set by hand. Nakhost and Müller (2009; 2013) introduce several global and local restart strategies.

Random Walk Length While the simplest approach is to use fixed length random walks, a better strategy in classical planning uses an *initial length bound*, and successively increases it if the best seen h-value does not improve quickly enough. If the algorithm encounters better states frequently enough, the length bound remains the same. A third strategy uses a *local restarting rate* to terminate a random walk with a fixed probability r_l after each motion. In this case, the length of walks is geometrically distributed with mean $1/r_l$.

Number of Random Walks The first version of Arvand used a fixed number of random walks in each search step, then progressed greedily to the best evaluated endpoint. This approach was later replaced by a number of adaptive methods (Nakhost and Müller 2009; 2013). A simple strategy followed here is to have only one random walk in a local search (Nakhost, Hoffmann, and Müller 2012), which is faster than choosing from among several walks, at the cost of solution quality.

Number of Search Episodes and Global Restarting The simplest global restart strategy restarts from initial state s_0 whenever the h-value fails to improve for a fixed number t_g of random walks. An *adaptive global restart* (AGR) algorithm is described in (Nakhost and Müller 2013).

Path Pool

Most versions of Arvand require very little memory. A *path pool* can store a number of random walks and utilize them for improving later searches (Nakhost, Hoffmann, and Müller 2012). The techniques of *On-Path Search Continuation* (OPSC) and *Smart Restarts* (SR) are based on a fixed-capacity pool which stores the most promising episodes encountered so far. OPSC randomly picks a state along the existing path to start a new search episode, instead of always starting from an endpoint. SR is used for global restart: instead of always restarting from s_0, the search restarts from a random state on a random path in the pool.

This current work only uses the path pool idea and pursued a different approach: to start a new search episode, a path p from the pool is either selected with the minimum h-value or randomly picked with a distribution; then a fixed fraction of the pool contents is replaced by newly generated random walks which extend p. Algorithm 3 shows details. The algorithm begins with an empty pool at each global (re-)start. A fixed number n, for example 10% of the pool size, is chosen for addition/replacement. n random walks are performed from start state s_0 and stored in the pool. During the search after (re-)start, one path in the pool is selected and expanded by local exploration to generate n new paths. If the pool is full, n randomly selected existing paths are replaced by new paths. Each path in the pool is a state sequence from s_0 to an endpoint s_j. If a solution is found during expansion, the plan is returned immediately.

Algorithm 3 Expand

Input current state s, goal state g, existing path p, number of new paths n, pool P
Output n new paths added to P, returns whether a solution was found

```
for n iterations do
   new_walk ← randomWalk(s, g)
   new_path ← p+ new_walk
   store(P, new_path)
   if solution found then
      return true
   end if
end for
return false
```

Bidirectional Arvand

Motion planners such as RRT and KPIECE have bidirectional variants with good performance. Bidirectional Arvand uses similar approach to solve planning problems. It maintains both a forward and a backward path pool. Explorations start from both the start state s_0 and a goal state g_0, and try to connect two search frontiers. For each pair of paths (p_f, p_b) in the two pools, the heuristic distance of their endpoints is stored. If the size of each pool is m, the time complexity of replacing n paths in the pool in each episode and updating the heuristic values is $O(nm)$.

Algorithm 4 shows the outline of bidirectional Arvand. In each search episode, search starts from the endpoint of one chosen path, treats the endpoint of the other chosen path as the search goal, and tries to connect them. In the code, $h(fPool, bPool) = \min_{f \in fPool, b \in bPool} h(f, b)$.

Algorithm 4 Bidirectional Arvand

Input current state s_0, goal state g_0, number of new paths n
Output A solution path

$h_{min} \leftarrow \infty$
init $\leftarrow$ *true*
repeat
  if *counter* $>$ *MAX_EPISODES* or *init* **then**
    counter $\leftarrow 0$
    fPool,*bPool* $\leftarrow \emptyset$
    $p \leftarrow NULL$
    expand(s_0, g_0, p, n, *fPool*)
    $s \leftarrow$ closest endpoint towards g_0 in *fPool*
    expand(s, s_0, p, n, *bPool*)
    current $\leftarrow$ *fPool*
    init $\leftarrow$ *false*
  end if
  reserve(n, *current*) {reserve room for n new paths}
  $s, g \leftarrow \text{argmin}_{f \in fPool, b \in bPool} h(f, b)$
  $p \leftarrow$ complete path towards s
  expand(s, g, p, n, *current*) {try to connect two paths}
  if h(*fPool*, *bPool*) $< h_{min}$ **then**
    $h_{min} \leftarrow h(fPool, bPool)$
    counter $\leftarrow 0$
  else
    counter $\leftarrow$ *counter* $+ 1$
  end if
  switch forward and backward search direction
until a solution is found
return solution path

Improving Plan Quality

The algorithms described above stop immediately after a solution is found. Arvand*, shown in Algorithm 5, is an optimizing version of Continuous Arvand, which keeps restarting even after the first valid plan is found. Arvand* uses post-processing techniques, such as shortcutting and smoothing, to simplify each newly found solution. The shortest solution after postprocessing is returned.

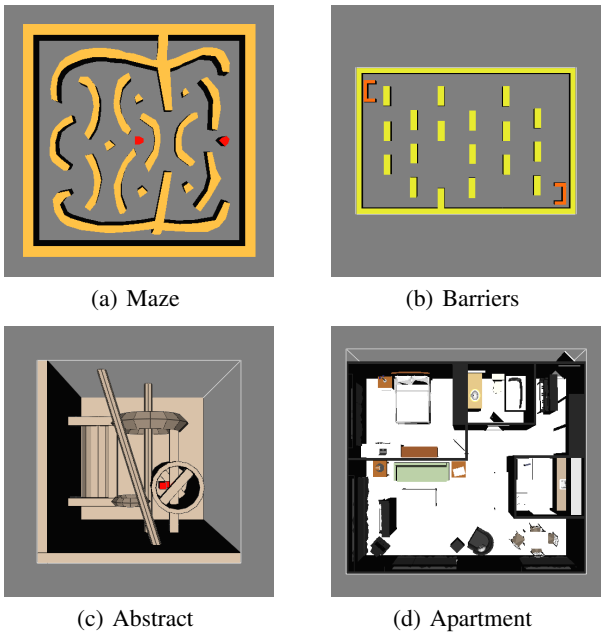
Algorithm 5 Arvand*

Input current state s_0, goal region G
Output A solution path with shortest length

$sol_{min} \leftarrow NULL$
while keep_going() **do**
  $sol \leftarrow$ monteCarloRandomWalk(s_0, G)
  $sol \leftarrow$ simplify(sol)
  if $sol_{min} = NULL$ **or** length(sol) $<$ length(sol_{min}) **then**
    $sol_{min} \leftarrow sol$
  end if
end while
return sol_{min}

3 Implementation - the Continuous Arvand System

Continuous Arvand implements a framework for MRW motion planning, and several different planners. The program is built on top of OMPL, the Open Motion Planning Library (Şucan, Moll, and Kavraki 2012). OMPL provides implementations of all motion planning primitives such as distance heuristics, collision detection, and random state sampling. The heuristic in Continuous Arvand is the distance function provided by OMPL, which differs depending on the type of state space. For instance, for state space $SO(3, \mathbb{R})$ the distance is the angle between quaternions, while $\mathbb{R}^3$ uses euclidean distance. The *simplify(path)* post-processing function provided by OMPL is used in all experiments to shorten the solutions.



(a) Maze (b) Barriers

(c) Abstract (d) Apartment

Figure 1: Planning scenarios

Six motion planners were implemented: Arvand_fixed and Arvand_extend are based on the techniques introduced

in (Nakhost and Müller 2009), while Arvand2 and Arvand2_AGR use ideas from (Nakhost and Müller 2013). BArvand and Arvand* are the bidirectional and optimizing variants of Arvand described in Section 2.

Arvand_fixed is the simplest implementation and uses constant parameters for global restart rate t_g, and number and length of random walks. In the experiments below, $t_g = 20$, *NUM_WALK* $= 20$, and *LENGTH_WALK* is tuned to find the best setting for each planning scenario, in the range from 10 to 800. Tuning these parameters is inconvenient and time consuming. The other versions of Arvand try to automatically adapt the setting for these variables. For Arvand_extend, *NUM_WALK* $= 800$, and *LENGTH_WALK* $= 10$ initially, and is multiplied by a factor of 2 whenever the h-value does not improve over 100 walks. In Arvand2, *NUM_WALK* $= 1$ and a local restarting rate of $r_l = 0.01$ is used to control the random walk length. Arvand2_AGR is similar to Arvand2, but adds adaptive global restarts. BArvand is the bidirectional version of Arvand. In experiments, the size of the forward and backward pools is 100 each, and the setting of other parameters is as in Arvand_fixed. Arvand* uses the same settings as Arvand2_AGR, but keeps running to improve solutions until a given time limit is reached.

4 Experiments

In this section, the five planners Arvand_fixed, Arvand_extend, Arvand2, Arvand2_AGR and BArvand are compared with a selection of the best-performing planners available in OMPL: RRT (LaValle and Kuffner 2001), KPIECE (Şucan and Kavraki 2010), EST (Hsu, Latombe, and Motwani 1997), PDST (Ladd and Kavraki 2005), and PRM (Kavraki et al. 1996). Arvand* is tested against RRT* (Karaman and Frazzoli 2011), which is an asymptotically-optimal incremental sampling-based motion planning algorithm.

Experiments used 13 built-in benchmark scenarios from OMPL: Maze, Barriers, Abstract, Apartment, BugTrap, RandomPolygons, UniqueSolutionMaze, Cubicles, Alpha, Easy, Home, Pipedream_ring and Spirelli. These scenarios are chosen as they can be solved by most available planners in reasonable time (less than 10 minutes). Four of them are shown in Figure 1. We grouped these scenarios into four categories: easy problems (Maze, BugTrap, RandomPolygons, Easy), intermediate problems (Alpha, Barriers, Apartment), intermediate problems with long detour (UniqueSolutionMaze, Cubicles, Pipedream_ring, Abstract) and hard problems (Home, Spirelli). The configuration space used in these problems is either $SE(2)$ or $SE(3)$. We used the recommended time limit provided in OMPL for each scenario in our experiments.

All experiments were run on a machine with 8-core CPU Intel Xeon E5420 @ 2.5GHz and 8GB memory. Results for each planner are averaged over 20 runs per scenario. The metrics of memory use (MB), path length, simplified path length, planning time and simplification time (in seconds) are considered.

Tables 3-16 show the benchmark results. For the metric of memory use, almost all Arvand versions always use less memory than all the other planners. One exception is the BArvand version in scenario Cubicles, which used more memory to maintain two path pools as this scenario has long detours, and BArvand needs much longer paths to reach the goal.

Considering the path length, Arvand2 and Arvand2_AGR always output solutions with huge path lengths. The reason is that these two Arvand versions do not run multiple random walks and choose the best one. Therefore they run faster but produce much longer paths. However, after post-processing, the simplified path length is good enough to compete with other planners. In scenarios Maze, RandomPolygons, Apartment and Easy, Arvand_fixed and Arvand_extend are comparable to other planners on original path length. In scenario Cubicles, these two planners are worse by a factor of 3 to 8. BArvand usually does not provide a competitive initial path length, but it performs very well after simplification. For instance, BArvand outperforms all other planners in scenarios Alpha and Barriers.

The total time in the experiments consists of planning time plus simplification time. The simplification time is insignificant: it is usually below 0.1s for all planners, and never reached 0.5s in any of the experiments. Therefore, only the total time is shown in the tables.

Among all Arvand versions, Arvand_fixed and Arvand_extend are slower because they run many random walks in one episode. This causes them to time out in scenarios UniqueSolutionMaze, Home and Spirelli. Arvand_fixed times out in more scenarios: Cubicles, Alpha, and Pipedream_ring. Arvand2 and Arvand2_AGR are competitive in terms of planning time for the easy planning problems Maze and BugTrap. They also do well in the intermediate problems Cubicles, Pipedream_ring and Apartment. BArvand performs well in most scenarios. It is the best planner in scenario Apartment, always takes a reasonable amount of time when comparing among all Arvand versions, and produces competitive short solutions.

For intermediate problems with long detours, almost all Arvand results are poor. The reason is that Arvand uses a heuristic to guide the exploration, and a detour requires the exploration to go multiple steps against the heuristic. Since Arvand2 and Arvand2_AGR only run one random walk per episode, the planning time is not bad. However, Arvand versions Arvand_fixed and Arvand_extend choose the heuristically best random walk among several walks. They have little chance to go against the heuristic for several successive steps, and need much more time to find a solution, or even time out. The only competitive results on these planning scenarios is for BArvand in scenario Abstract.

The performance of the optimizing planners RRT* and Arvand* within the recommended time limit is shown in Table 16. On the main metric of simplified path length, RRT* is always better than Arvand*, and Arvand* comes close to the performance of RRT* only in scenarios Alpha and Easy.

The picture can change with longer time limits. Figure 2 compares the plan improvement over time for the two planners in scenario Alpha. While the initial solution found by Arvand* is of rather poor quality, its path length decreases rapidly over time and becomes better than RRT* in this ex-

ample. At the current time, this is an isolated positive result and it is not clear whether it generalizes to other scenarios.

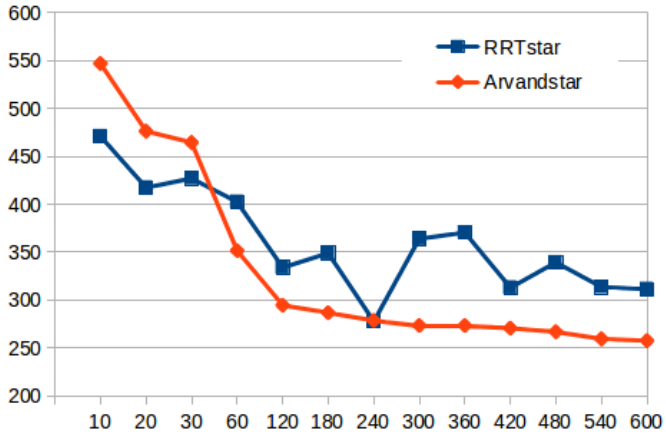


Figure 2: Plan improvement over time for Arvand* and RRT*. Average over 10 runs.

For generating Figure 2, since the intermediate paths when RRT* is optimizing its plan are not accessible, RRT* is run separately for different time limits. Each data point is averaged over 10 runs.

As a final example, Table 2 shows the importance of choosing the right parameters for MRW with fixed settings. The example is from scenario Barriers, as solved by Arvand_fixed with different parameter settings. For setting 1, the length of random walks is 20, the number of walks per episode is 20, and the maximum number of episodes is 10; for setting 2, the length of random walks is 1000, the number of walk per episode is 50, and the maximum number of episodes is 100. In this planning scenario, setting 1 has better performance.

	Solution Path	Planning Time
Setting 1	1205.7	8.7s
Setting 2	2063.8	24.1s

Table 2: Influence of parameter setting on performance.

5 Conclusions and Future Work

The algorithms developed in this paper apply the Monte Carlo Random Walk method to motion planning. Global and local restart strategies in this method have huge impact on performance. Our work is still preliminary, but the results are already interesting. Continuous Arvand works well for problems that do not require long detours for which the distance heuristic is misleading. The algorithms use much less memory than other planners, which makes them attractive for embedded applications with limited resources.

Portfolio planning (Gomes and Selman 2001) combines several algorithms into a portfolio and runs them in sequence or in parallel. This is a very successful approach in classical planning. The *ArvandHerd* system, winner of the parallel satisficing track of the 2011 and 2014 International Planning Competitions, is such a portfolio which combines (classical) Arvand with another state of the art planner, LAMA (Valenzano et al. 2012; 2014). Our results indicate that adding Continuous Arvand to a motion planning portfolio will very likely strengthen its performance.

The current versions of Continuous Arvand do not work well on planning problems with long forced detours that go against the heuristic. Improving the performance on these kinds of problems is the most important task for future work. Some existing MRW techniques from classical planning, such as On-Path Search Continuation (OPSC) and Smart Restarts (SR) (Nakhost, Hoffmann, and Müller 2012), are not yet used in the Continuous Arvand implementation. *Adaptive local restarting* (Nakhost and Müller 2013) is a technique used to estimate the best parameter for local restarting. In addition of only evaluating the endpoint of a random walk, Arvand could benefit from the heuristic evaluation of the intermediate states along the walk (Nakhost, Hoffmann, and Müller 2012). Finally, there is work to do to research the many different ways of using memory, such as different strategies for using path pools, adding a tree as in RRT, or a UCT-like approach.

References

Gomes, C. P., and Selman, B. 2001. Algorithm portfolios. *Artificial Intelligence* 126(1):43–62.

Hsu, D.; Latombe, J.-C.; and Motwani, R. 1997. Path planning in expansive configuration spaces. In *IEEE Robotics and Automation*, volume 3, 2719–2726. IEEE.

Karaman, S., and Frazzoli, E. 2011. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research* 30(7):846–894.

Kavraki, L. E.; Svestka, P.; Latombe, J.-C.; and Overmars, M. H. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Robotics and Automation* 12(4):566–580.

Ladd, A. M., and Kavraki, L. E. 2005. Motion planning in the presence of drift, underactuation and discrete system changes. In *Robotics: Science and Systems*, 233–240.

LaValle, S. M., and Kuffner, J. J. 2001. Randomized kinodynamic planning. *The International Journal of Robotics Research* 20(5):378–400.

LaValle, S. M. 2006. *Planning algorithms*. Cambridge University Press.

Nakhost, H., and Müller, M. 2009. Monte-Carlo exploration for deterministic planning. In *IJCAI*, volume 9, 1766–1771.

Nakhost, H., and Müller, M. 2013. Towards a second generation random walk planner: an experimental exploration. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, 2336–2342. AAAI Press.

Nakhost, H.; Hoffmann, J.; and Müller, M. 2012. Resource-constrained planning: A Monte Carlo random walk approach. In McCluskey, L.; Williams, B.; Reinaldo Silva, J.; and Bonet, B., eds., *ICAPS*, 181–189. AAAI Press.

Şucan, I. A., and Kavraki, L. E. 2010. Kinodynamic motion planning by interior-exterior cell exploration. In *Algorithmic Foundation of Robotics VIII*. Springer. 449–464.

Şucan, I. A.; Moll, M.; and Kavraki, L. E. 2012. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine* 19(4):72–82. `http://ompl.kavrakilab.org`.

Valenzano, R.; Nakhost, H.; Müller, M.; Sturtevant, N.; and Schaeffer, J. 2012. ArvandHerd: Parallel planning with a portfolio. In De Raedt, L., ed., *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, 786–791. IOS Press.

Valenzano, R.; Nakhost, H.; Müller, M.; Schaeffer, J.; and Sturtevant, N. 2014. Arvandherd 2014. In Vallati, M.; Chrpa, L.; and McCluskey, T., eds., *The Eighth International Planning Competition*, 1–5. University of Huddersfield.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	1.26	285.35	149.64	**0.49**
EST	2.27	189.72	118.11	1.58
PDST	16.64	195.17	117.50	0.59
RRT	0.89	152.16	125.07	0.59
PRM	1.64	134.95	116.70	1.10
Arvand_fixed	0.36	**120.68**	**88.72**	5.39
Arvand_extend	0.47	187.00	105.30	7.16
Arvand2	0.98	4,630.43	139.96	1.74
Arvand2_AGR	2.04	10,739.10	153.31	1.75
BArvand	**0.52**	364.63	108.33	0.70

Table 3: Scenario Maze, time limit $= 20s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	2.71	3,410.69	1,738.58	**0.61**
EST	6.11	2,130.88	1,544.93	2.37
PDST	29.83	3,058.34	2,078.17	0.91
RRT	243.23	1,723.06	1,519.71	1.20
Arvand_fixed	**0.40**	**1,468.52**	1,075.18	15.93
Arvand_extend	0.70	4,253.87	1,416.57	28.52
Arvand2	1.65	35,791.87	1,623.66	4.10
Arvand2_AGR	3.14	111,872.24	1,714.82	3.49
BArvand	4.48	7,690.83	**864.12**	5.36

Table 4: Scenario Barriers, time limit $= 300s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	19.87	3,180.60	1,070.92	14.53
EST	17.18	1,567.90	855.60	16.59
PDST	199.83	2,764.19	1,228.19	14.71
RRT	153.39	1,256.63	949.70	29.57
PRM	160.16	**805.39**	706.16	258.84
Arvand_fixed	0.88	1,388.88	647.05	166.39
Arvand_extend	2.05	10,127.76	887.14	96.83
Arvand2	2.46	23,285.37	786.86	133.29
Arvand2_AGR	21.16	904,221.82	998.34	36.69
BArvand	**0.59**	1,395.73	**589.79**	**11.02**

Table 5: Scenario Abstract, time limit $= 300s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	10.43	1,133.78	452.93	17.25
EST	3.79	716.99	444.24	12.95
PDST	92.36	920.69	437.46	19.85
RRT	3.35	523.93	425.96	**8.30**
PRM	50.04	**485.74**	**409.94**	102.30
Arvand_fixed	**0.36**	529.79	428.66	38.09
Arvand_extend	0.52	859.32	437.66	96.77
Arvand2	0.63	2,859.56	431.09	9.17
Arvand2_AGR	0.72	4,233.13	458.73	12.35
BArvand	2.41	2,436.83	445.05	10.78

Table 6: Scenario Apartment, time limit $= 300s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	4.62	446.65	170.58	0.33
EST	2.73	286.95	162.31	0.41
PDST	17.22	303.57	175.14	**0.26**
RRT	2.50	254.89	177.31	0.33
PRM	9.73	**163.42**	**140.59**	3.42
Arvand_fixed				time out
Arvand_extend	**0.79**	867.29	165.84	4.94
Arvand2	1.91	10,514.92	167.31	0.73
Arvand2_AGR	2.78	16,058.34	160.42	0.77
BArvand	3.74	1,210.20	162.08	1.28

Table 7: Scenario BugTrap, time limit $= 20s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	0.94	310.00	133.65	0.11
EST	0.75	228.59	130.62	0.31
PDST	1.60	196.65	133.00	0.06
RRT	0.57	155.47	130.49	**0.05**
PRM	0.55	**149.82**	133.57	0.10
Arvand_fixed	**0.34**	189.09	**123.66**	4.20
Arvand_extend	0.43	290.43	127.64	2.26
Arvand2	0.79	2,192.15	137.73	0.18
Arvand2_AGR	0.86	2,137.46	132.41	0.14
BArvand	0.63	404.06	124.78	0.21

Table 8: Scenario RandomPolygons, time limit $= 20s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	3.29	663.00	393.32	**1.33**
EST	18.46	491.09	367.62	3.71
PDST	202.19	483.52	337.94	5.63
RRT	2.99	399.27	344.27	2.77
PRM	3.09	**340.60**	**328.99**	2.31
Arvand_fixed				time out
Arvand_extend				time out
Arvand2	**2.55**	8,134.98	346.50	6.66
Arvand2_AGR	6.13	42,158.52	341.17	7.73
BArvand	7.81	1,092.37	352.22	4.63

Table 9: Scenario UniqueSolutionMaze, time limit $= 20s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	6.25	6,592.95	2,606.57	1.14
EST	10.96	3,888.98	2,450.79	6.81
PDST	55.02	4,805.96	2,555.92	2.41
RRT	**0.91**	3,242.16	2,587.69	**0.60**
PRM	12.34	**2,512.20**	**2,292.76**	5.33
Arvand_fixed				time out
Arvand_extend	1.14	20,054.38	2,481.16	46.52
Arvand2	2.16	61,197.39	2,442.39	1.95
Arvand2_AGR	3.01	85,836.54	2,423.80	1.81
BArvand	48.50	34,533.70	2,454.40	7.59

Table 10: Scenario Cubicles, time limit $= 60s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	**0.61**	1,614.49	637.16	2.79
EST	0.78	**938.13**	550.47	4.12
PDST	19.65	1,569.88	576.56	**2.04**
RRT	17.15	949.06	583.25	4.12
PRM				time out
Arvand_fixed				time out
Arvand_extend	3.19	2,000.73	496.25	25.07
Arvand2	3.45	17,592.26	563.81	8.75
Arvand2_AGR	3.67	19,263.26	622.31	6.12
BArvand	3.45	6,947.17	**481.80**	18.85

Table 11: Scenario Alpha, time limit $= 60s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	2.41	1,140.28	234.30	0.63
EST	2.94	593.66	236.81	0.64
PDST	6.05	595.02	233.84	0.17
RRT	8.41	**347.84**	209.43	**0.15**
PRM	8.63	508.85	250.27	0.46
Arvand_fixed	**0.39**	369.39	205.96	0.56
Arvand_extend	0.43	594.30	**204.28**	0.73
Arvand2	0.92	33,561.37	216.11	1.19
Arvand2_AGR	2.49	126,037.08	208.22	1.10
BArvand	4.48	8,522.32	236.58	0.90

Table 12: Scenario Easy, time limit $= 20s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE				time out
EST	**303.22**	4,127.54	2,364.86	31.00
PDST	3,229.55	3,955.31	1,908.11	128.53
RRT	3,581.81	2,585.87	2,141.99	**26.56**
PRM	3,584.59	**1,825.84**	**1,637.01**	59.76
Arvand_fixed				time out
Arvand_extend				time out
Arvand2				time out
Arvand2_AGR				time out
BArvand				time out

Table 13: Scenario Home, time limit $= 300s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE	21.54	242.65	98.14	62.14
EST	19.45	161.73	89.26	2.52
PDST	22.73	241.60	108.95	**1.03**
RRT	23.45	157.41	105.98	2.02
PRM	223.71	**128.58**	**86.23**	90.12
Arvand_fixed				time out
Arvand_extend	**2.91**	385.03	131.80	75.86
Arvand2	3.18	1,564.64	116.08	1.53
Arvand2_AGR	3.30	1,862.63	104.08	1.42
BArvand	3.97	957.35	133.25	14.39

Table 14: Scenario Pipedream_ring, time limit $= 300s$.

Planner	Memory	Path length	Simplified path length	Total time
KPIECE				time out
EST				time out
PDST				time out
RRT	2,229.50	**203.22**	**166.05**	102.99
PRM				time out
Arvand_fixed				time out
Arvand_extend				time out
Arvand2				time out
Arvand2_AGR	1.41	3,043.27	222.48	**78.68**
BArvand	**0.51**	480.48	166.80	157.13

Table 15: Scenario Spirelli, time limit $= 180s$.

Problem	Planner	Memory	Path length	Simplified path length
Alpha	RRTstar	324.36	358.81	328.52
	Arvandstar	37.63	5,890.36	358.97
Barriers	RRTstar	510.36	822.51	806.70
	Arvandstar	120.97	51,703.69	1,294.87
Easy	RRTstar	266.48	208.19	203.44
	Arvandstar	55.46	62,327.05	211.09
Pipedream_ring	RRTstar	471.33	84.77	77.14
	Arvandstar	82.89	375.72	99.38
Spirelli	RRTstar	8.52	118.59	114.89
	Arvandstar	0.70	2,877.63	193.00
Abstract	RRTstar	236.63	600.64	569.85
	Arvandstar	298.18	438,366.05	862.49
Apartment	RRTstar	57.75	404.29	382.32
	Arvandstar	4.42	2,062.75	432.15
BugTrap	RRTstar	29.55	124.66	121.78
	Arvandstar	51.72	7,040.70	163.44
Cubicles	RRTstar	18.38	1,916.82	1,833.96
	Arvandstar	60.56	45,079.29	2,378.78
Maze	RRTstar	9.71	71.51	69.69
	Arvandstar	19.45	829.89	102.78
RandomPolygons	RRTstar	16.51	108.78	106.53
	Arvandstar	37.21	646.64	127.31
UniqueSolutionMaze	RRTstar	11.35	280.85	277.49
	Arvandstar	15.93	25,511.39	345.38

Table 16: Comparing RRT* and Arvand*, with the same time limits as in previous tables.

Frontier-Based RTDP: A New Approach to Solving the Robotic Adversarial Coverage Problem

Roi Yehoshua, Noa Agmon and Gal A. Kaminka
Computer Science Department
Bar Ilan University, Israel
{yehoshr1,agmon,galk}@cs.biu.ac.il

Abstract

Area coverage is an important problem in robotics, where one or more robots are required to visit all points in a given area. In this paper we consider a recently introduced version of the problem, *adversarial coverage*, in which the covering robot operates in an environment that contains threats that might stop it. The objective is to cover the target area as quickly as possible, while minimizing the probability that the robot will be stopped before completing the coverage. We first model this problem as a Markov Decision Process (MDP), and show that finding an optimal policy of the MDP also provides an optimal solution to this problem. Since the state space of the MDP is exponential in the size of the target area's map, we use real-time dynamic programming (RTDP), a well-known heuristic search algorithm for solving MDPs with large state spaces. Although RTDP achieves faster convergence than value iteration on this problem, practically it cannot handle maps with sizes larger than 7×7. Hence, we introduce the use of frontiers, states that separate the covered regions in the search space from those uncovered, into RTDP. Frontier-Based RTDP (FBRTDP) converges orders of magnitude faster than RTDP, and obtains significant improvement over the state-of-the-art solution for the adversarial coverage problem.

Introduction

There are many real-life applications that require a robot, or a group of robots, to cover an area. For example, a vacuum cleaning robot that needs to clean an entire room (Colegrave and Branch 1994), intrusion detection, mine cleaning (Nicoud and Habib 1995) and search-and-rescue missions.

Most previous studies of the coverage problem dealt with non-adversarial settings, where nothing in the environment is hindering the robot's task. However, on many occasions, robots and autonomous agents need to perform coverage missions in hazardous environments, such as operations in nuclear power plants, exploration of Mars, demining and search and rescue in the battlefield.

Hence, our work addresses the problem of planning for a robot whose task is to cover a given terrain without being detected or damaged by an adversary, as introduced in (Yehoshua, Agmon, and Kaminka 2013). Each point in the area is associated with a probability of the robot being stopped at that point. The objective of the robot is to cover the *entire* target area as quickly as possible while maximizing its own safety. We will refer to this problem as the *adversarial coverage problem*. Here we discuss the offline version of this problem, in which the map of threats is given in advance, therefore the coverage path of the robot can be determined prior to its movement.

In this paper, we show how the adversarial coverage problem can be modeled as a Markov Decision Process (MDP). MDPs are a natural choice for implementing a solution to this problem because they explicitly represent costs and uncertainty in results of actions, as well as doing lookahead to examine the potential consequences of sequences of actions. We show that, given an appropriate definition of the MDP, finding an optimal policy for the model also provides an optimal solution to the adversarial coverage problem.

Since the state space of the MDP is exponential in the size of the target area's map, using classical (synchronous) value iteration techniques to find the optimal policy is possible only for small-sized maps. Therefore, in order to handle larger maps, we use real-time dynamic programming (RTDP), which is a well-known heuristic search algorithm for solving MDPs with intractably large state space (Barto, Bradtke, and Singh 1995). Although RTDP achieves faster convergence than value iteration on this problem, practically it cannot handle maps with sizes larger than 7×7. This is because the search graph representing the problem is highly connected, and thus RTDP trials often get trapped in loops (moving repeatedly between the same states).

Hence, we introduce the use of frontiers, states that separate the covered regions in the search space from those uncovered, into RTDP. In each step of the trial, Frontier-Based RTDP (FBRTDP) searches for a path with minimum expected cost from the current state to a new frontier state, choosing outcomes of this path stochastically according to their probability. FBRTDP avoids getting trapped in loops by advancing towards a new frontier in each step. The use of frontiers speeds up the convergence of RTDP quite dramatically, while retaining its focus and anytime behavior. Finally, we show that FBRTDP attains significant improvement over the state-of-the-art solution to the problem (Yehoshua, Agmon, and Kaminka 2014), in terms of both the coverage time and probability to complete the coverage.

Related Work

The problem of robot coverage has been extensively discussed in the literature (see (Galceran and Carreras 2013) for a recent exhaustive survey). Grid-based coverage methods, such as we utilize here, use a representation of the environment decomposed into a collection of uniform grid cells, e.g., (Gabriely and Rimon 2003), (Luo et al. 2002).

Other optimization problems related to adversarial coverage include the Canadian Traveller Problem (CTP) (Papadimitriou and Yannakakis 1989), in which the objective is to find a shortest path between two nodes in a partially-observable graph, where some edges may be non-traversable. In contrast, here the graph is fully-observable and the agent must visit every node in the graph (some of them may stop the robot). MDPs have been shown to be useful in solving CTP in certain types of graphs (Nikolova and Karger 2008).

The offline adversarial coverage problem was formally defined in a recent study (Yehoshua, Agmon, and Kaminka 2013). There the authors proposed a simplistic heuristic algorithm that generates a coverage path which tries to minimize a cost function, that takes into account both the survivability of the robot and the coverage path length. The heuristic algorithm worked only for obstacle-free areas, and without any guarantees. In a follow-up paper (Yehoshua, Agmon, and Kaminka 2014) we have addressed a more specific version of the problem, namely, finding the safest coverage path. We suggested two heuristic algorithms to solve this problem with some theoretical guarantees. However, they could handle only one level of threats, i.e., the environment could contain either safe or dangerous areas. In contrast, here we suggest a model that can find optimal (or near optimal) coverage paths that meet any desired risk and time levels in environments that can contain any number of threat levels, in addition to obstacles.

Perhaps the simplest algorithm for solving MDPs is value iteration, which solves for an optimal policy on the full state space. However, in many realistic problems, such as the one we discuss here, only a small fraction of the state space is relevant to the problem of reaching a goal state from a fixed start state s. There are different heuristic algorithms for solving MDPs, including offline and real-time methods. Real-Time Dynamic Programming (Barto, Bradtke, and Singh 1995) and its variants (such as Labeled RTDP (Bonet and Geffner 2003)) have been shown to outperform other heuristic search methods for solving MDPs, such as AO*/LAO*, on several benchmark problems (Bonet and Geffner 2003). The advantage of real-time heuristic methods is that states that are more likely to be visited in the search graph (as defined by the probability function) are updated more frequently, which leads the algorithm to focus on updating states which are more relevant to the problem solving. In our case, states that are more likely to be visited by the robot represent coverage paths that encounter a smaller number of threats, thus updating those states more frequently can focus the search for the optimal coverage path.

Adversarial Coverage Problem Definition

We are given a map of a target area T, which is decomposed into a regular square grid with n cells. We assume that T can be decomposed into a regular square grid with n cells. There are two types of cells: free cells and cells that are occupied by obstacles. Some of the free cells contain threats. Each free cell c_i is associated with a threat probability p_i ($0 \leq p_i < 1$), which measures the likelihood that a threat in that cell will stop the robot. We assume the robot can move continuously, in the four basic directions (up/down, left/right), and can locate itself within the work-area to within a specific cell. The robot's task is to plan a path through T such that every accessible free cell in T (including the threat points) is visited by the robot at least once.

The objective is to find a coverage path of T that maximizes the probability of covering the entire area and also minimizes the coverage time. Clearly, there is a tradeoff between these two objectives: trying to minimize the risk involved in the coverage path could mean making some redundant steps, which in turn can make the coverage path longer, and thus increase the risk involved, as well as increase the coverage time.

We now formally define this objective function. First, we denote the coverage path followed by the robot by $A = (a_1, a_2, ..., a_m)$. Note that $m \geq n$, i.e., the number of cells in the coverage path might be greater than the number of cells in the target area, since the robot is allowed to repeat its steps. We define the event S_A as the event that the robot is not stopped when it follows the path A. The probability that the robot is able to complete this path is:

$$P(S_A) = \prod_{i \in (a_1,...,a_m)} (1 - p_i) \tag{1}$$

In order to take into account both the accumulated risk and the coverage time, we define the following cost function:

$$f(A) = -\alpha \cdot P(S_A) + \beta \cdot |A| \tag{2}$$

where $\alpha, \beta \geq 0$, and $|A|$ is the number of the steps the robot needs to take in order to complete the coverage path. [1] Therefore, we wish to find a coverage path A that minimizes the cost function $f(A)$, i.e., $f(A) \leq f(B)$ for all possible coverage paths B.

The problem of finding a minimum time coverage path ($\alpha = 0$) is equivalent to finding a Hamiltonian walk in a graph, which is known to be $\mathcal{NP}$-complete (Nishizeki, Asano, and Watanabe 1983). Finding a coverage path with maximum probability to complete ($\beta = 0$) has also been shown to be $\mathcal{NP}$-complete (Yehoshua, Agmon, and Kaminka 2014). By reduction, the problem in the general case of $\alpha, \beta \geq 0$ is $\mathcal{NP}$-complete as well.

MDP Modeling

We now formulate the adversarial coverage problem as an undiscounted stochastic shortest-path problem (Bertsekas 1995). Stochastic shortest path (SSP) problems are a subclass of MDPs, and they are given by:

[1] The travel time is uniform along the grid, thus coverage time is measured directly by the number of steps.

S1. a discrete and finite state space S
S2. an initial state $s_0 \in S$
S3. a set $G \subseteq S$ of goal states
S4. Actions $A(s) \subseteq A$ applicable in each state $s \in S$
S5. Transition probabilities $P(s'|s,a)$ for $s \in S, a \in A(s)$
S6. Positive action costs $C_a(s,s') > 0$
S7. Fully observable states

Let us denote the SSP that represents the adversarial coverage problem by $\mathcal{M}$. We now describe each of $\mathcal{M}$'s components.

States. The set of states in our model contains all possible configurations of the environment's coverage status and the robot's location. A coverage status of the environment is represented by a boolean matrix that indicates for each cell in the grid if it has already been visited by the robot or not. The state captures all relevant information from the history of the robot's movements, thus it satisfies the Markovian property.

The initial state s_0 is the state in which the robot is located at the starting cell of the coverage path, and this is the only cell marked as visited. The goal states G are the states in which all the grid cells are covered. In addition, one of the states in S is defined as a dead state, denoted by s_d, which represents the situation where the robot was stopped by a threat. This dead-end state requires special attention, as discussed in section . The goal states, as well as s_d, are termination states, i.e., taking any action in them causes a self-transition with probability 1.

Actions. There are only four possible actions the robot can perform - go up, down, left or right. There could be fewer than four actions applicable in a given state, depending on the number of obstacles that surround the cell in which the robot currently resides.

The Transition Function. The transition function describes the probability that the robot will be able to move from its current location to the next location on its coverage path. More specifically, if the current state is s, in which the robot is located in cell c_i, and the robot executes an action a that leads it to cell c_j, then we distinguish between two possible cases:

Case 1. c_j is a safe cell, i.e., $p_j = 0$. In this case there is only one possible outcome for (s,a). The outcome is a new state s', in which c_j is added to the environment's coverage status and the robot's location is changed to cell c_j. The transition probability in this case is $P_a(s'|s) = 1$.

Case 2. c_j is a dangerous cell. In this case there are two possible successor states to (s,a). The first one, s', represents the possibility that the robot will be able to enter the new cell c_j. In s', c_j is added to the environment's coverage status and the robot's location is changed to cell c_j. The transition probability to s' is $P_a(s'|s) = 1 - p_j$, where p_j denotes the probability that the threat in cell c_j will stop the robot. The second successor state is s_d, which represents the possibility that the robot will be stopped by a threat in the cell c_j. The transition probability to this state is $P_a(s_d|s) = p_j$.

Note that the probabilities on all the outgoing transitions of each state/action pair sum to 1.

The Cost Function. We define a uniform fixed cost $C_a(s,s') = 1$ for actions that lead the robot to a safe cell.

For actions that lead the robot to a cell c_j with threat probability p_j, we define different costs for the two possible outcomes of this action. If the robot was not hit by the threat, the cost of the transition to the next state s' is defined as $C_a(s,s') = \frac{1}{1-p_j}$. Otherwise, the transition cost to the dead-end state s_d is defined as $C_a(s,s_d) = -D \cdot \frac{\log(1-p_j)}{p_j}$, where $D \geq 0$ is a fixed penalty assigned to reaching the dead state.

The value of the penalty D should be set according to the desired balance between the risk and the coverage time (α/β). For example, setting $D = 0$ should make the model find the shortest coverage path, while setting $D = \infty$ should make it find the safest coverage path. For the in-between cases, to help us calibrate the value of D, we will define that when $\alpha = \beta$, i.e., when the risk and the coverage time factors have equal importance, the penalty on making a move to a dangerous cell (with a minimum threat probability) will be equal to making one step in the grid. In particular, if p_{min} is the minimum threat probability, then D is set to:

$$D = -\frac{\alpha}{\beta} \cdot \frac{1}{\log(1-p_{min})} \tag{3}$$

Note that this is the only place in the model where the risk and the time factors are combined together.

This concludes the definition of the model $\mathcal{M}$ representing the adversarial coverage problem. To demonstrate the model, let us consider the following simple grid (cells are numbered 1 to 2 from top to bottom and left to right, the numbers in the cells indicate the threat probabilities p_i):

0	0
0.4	0.2

Assume that the robot starts the coverage at cell $(1,1)$ and then moves right to cell $(1,2)$. Let us denote the current state of the environment and the robot by s_1. Figure 1 shows the graph describing the possible transitions from s_1. Circular nodes of the graph represent states of the MDP and the rectangular nodes represent actions. Inside each state node there is a description of the coverage status of the environment and the robot's position (marked by 'R'). Edges from actions to states are annotated with transition probabilities and costs.

As can be seen in the graph, there are two possible actions in state s_1: going left (a_1) and going down (a_2). Moving left to the safe cell $(1,1)$ (i.e., choosing action a_1) has only one possible outcome with probability 1. In the resultant state s_2, the coverage status of the environment does not change, only the robot's location. On the other hand, going down to the dangerous cell $(2,2)$ (i.e., choosing action a_2) leads to two possible outcomes. If the move succeeds, i.e., the robot is not stopped by the threat, then the state changes to s_3, in which the robot is located at cell $(2,2)$ and this cell is added to the coverage status of the environment. The cost of this transition is: $\frac{1}{1-0.2} = 1.25$. However, if the move fails, then the robot moves to the dead-end state s_d. The probability of the transition to s_d is 0.2, which is the threat probability in cell $(2,2)$. The cost of this transition

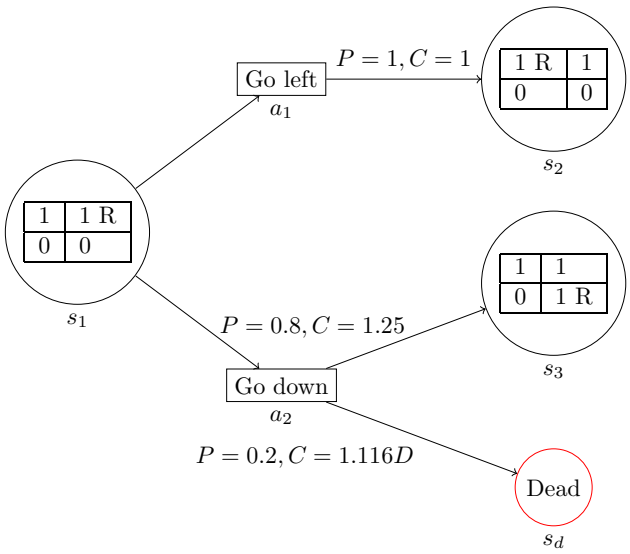


Figure 1: An example for a state in the MDP and its outgoing transitions. Edges from actions to states are annotated with transition probabilities and costs.

is: $-D \cdot \frac{\log(1-0.2)}{0.2} = 1.116D$.

The solution of an MDP takes the form of a *policy* π mapping states s into actions $a \in A(s)$. The value function of a policy π, V^π, represents the expected cost incurred from following policy π from any given state s in S, i.e.:

$$V^\pi(s) = E\left[\sum_{t=0}^{\infty} C_{a_t}(s_t, s_{t+1})\right] \quad (4)$$

where $s_0 = s$, and $a_t = \pi(s_t)$ is the action taken at time step t and causes a transition from state s_t to state s_{t+1}.

An *optimal policy* is a policy π^* that has a minimum expected cost for all possible initial states. Such a policy is guaranteed to exist if the following assumption holds (Bertsekas 1995):

S8. The goal is reachable from every state with non-zero probability.

In $\mathcal{M}$, all states except for the dead-end state satisfy this assumption (since all threat probabilities are less than 1). We discuss how to treat the dead-end state in section .

An optimal value function, denoted by $V^*(s)$, assigns to each state its value according to an optimal policy π^*, and satisfies the following fixed point equation, also known as Bellman's optimality equation (Bellman 1957):

$$V^*(s) = \min_{a \in A(s)} \sum_{s' \in S} P(s'|s,a)\big[C_a(s,s') + V^*(s')\big] \quad (5)$$

Value iteration (VI) is a standard dynamic programming method for solving MDPs based on Eq. (5). VI algorithms start with an initial guess for V_0 and repeatedly update so that V gets closer to V^*.

We now prove the correctness of the model $\mathcal{M}$.

Theorem 1. *(correctness) The optimal policy of the MDP $\mathcal{M}$ represents an optimal solution to the adversarial coverage problem.*

Proof. By definition, the optimal policy π^* of MDP $\mathcal{M}$ minimizes the expected cost:

$$E\left[\sum_{t=0}^{\infty} C_{a_t}(s_t, s_{t+1})\right] \quad (6)$$

where s_0 is the initial state, and $a_t = \pi^*(s_t)$ is the action taken at time step t according to the policy π^*.

The sequence of actions $(a_0, a_1, a_2, ...)$ taken by the optimal policy must eventually lead to a goal state. This is due to the fact that all action costs are positive (except for actions in the goal states), thus if the sequence of actions never reaches a goal, then the expected cost of the optimal policy becomes infinite, which violates assumption S8 that states that there must be at least one policy that reaches a goal state from any state.

Now denote by n the number of state transitions needed for the optimal policy to reach a goal state from the initial state s_0. Then the expected cost can be written as:

$$E\left[\sum_{t=0}^{n-1} C_{a_t}(s_t, s_{t+1})\right] \quad (7)$$

where s_n is a goal state.

From the linearity of expectation, we get:

$$E\left[\sum_{t=0}^{n-1} C_{a_t}(s_t, s_{t+1})\right] = \sum_{t=0}^{n-1} E\big[C_{a_t}(s_t, s_{t+1})\big] \quad (8)$$

Now let us denote by $(c_1, ..., c_n)$ the sequence of cells that were visited by the robot following the actions in $(a_0, ..., a_{n-1})$, and their threat probabilities by $(p_1, ..., p_n)$. According to the cost function $C_a(s, s')$ in the MDP model, the expected cost of moving to cell c_j is:

$$\begin{aligned} E[C(c_j)] &= (1 - p_j) \cdot \frac{1}{1 - p_j} + p_j \cdot \left[- D \cdot \frac{\log(1 - p_j)}{p_j} \right] \\ &= 1 - D \log(1 - p_j) \end{aligned} \quad (9)$$

The resultant expression is also true for the expected cost of moving to a safe cell c_j, since in that case $p_j = 0$, thus $E[C(c_j)]$ becomes equal to 1.

Thus, the sum in Eq. (8) becomes:

$$\sum_{t=0}^{n-1} E\big[C_{a_t}(s_t, s_{t+1})\big] = n - D \sum_{j=1}^{n} \log(1 - p_j) \quad (10)$$

The result is a sum of two expressions - the first is determined by the coverage path length (n) and the second is determined by the accumulated risk that was taken by the robot along the path. It is trivial to verify that if there are two coverage paths with the same accumulated risk but with different lengths, then the optimal policy will prefer the shorter

one. We now prove that if there are two coverage paths with the same length but with different accumulated risks, then the optimal policy will prefer the safer path. Let us denote the sequence of cells visited along the first coverage path by $(u_1, ..., u_l)$ and the sequence of cells visited along the second path by $(v_1, ..., v_m)$. Let us assume that the first path is safer than the second, i.e., it has a greater probability to complete. Thus, we can write:

$$\prod_{i\in(u_1,...,u_l)} (1-p_i) \geq \prod_{j\in(v_1,...,v_m)} (1-p_j) \qquad (11)$$

Since the logarithm is a monotonically increasing function of its argument, the above expression is equivalent to:

$$\sum_{i\in(u_1,...,u_l)} \log(1-p_i) \geq \sum_{j\in(v_1,...,v_m)} \log(1-p_j) \qquad (12)$$

If we multiply both sides by $-D$ and add n (the path length), we get:

$$n-D \sum_{i\in(u_1,...,u_l)} \log(1-p_i) \leq n-D \sum_{j\in(v_1,...,v_m)} \log(1-p_j) \qquad (13)$$

Note that the expressions on both sides of Eq. (13) are similar to the expression on the right-hand side of Eq. (10). Thus, we can conclude that the expected cost of a policy that generates the first coverage path is lower than the expected cost of a policy that generates the second one. Therefore, the optimal policy of MDP $\mathcal{M}$ is guaranteed to produce the safer coverage path.

In the in-between cases, where we want to find an optimal coverage path that takes into account both the coverage time and the accumulated risk, we can adjust the penalty D according to the desired levels of risk and the time. The optimal policy will then produce a coverage path that minimizes the expected cost defined in Eq. (10), which is dependent on D. The higher the penalty D is set, the more safer coverage paths will be favored over shorter ones. □

MDPs with Dead Ends

Researchers have realized that allowing dead ends in goal-oriented MDPs could break the existing methods for solving them (e.g., (Little and Thiebaux 2007)). In MDPs with dead ends the objective of finding a policy that minimizes the expected cost of reaching the goal becomes ill-defined, since it implicitly assumes that for at least one policy the cost incurred by all of the policy's trajectories is finite.

This problem can be resolved by assigning a finite positive penalty D for visiting a dead end, and augmenting the action set A of the MDP with a special action a' that causes a transition from the dead end to the goal with probability 1. This MDP now satisfies assumption S8, since reaching the goal with certainty is possible from every state. However, this solution comes with a caveat - it may cause non-dead-end states that lie on potential paths to a dead end to have higher costs than the dead ends themselves. As a consequence, the optimal policy may prefer getting into a dead end rather than reaching the goal. Kolobov et al. (Kolobov, Mausam, and Weld 2012) suggest resolving this issue by capping the cost of each state by D. They use the following modified Bellman equation:

$$V^\pi(s) = \min\Big\{D, \min_{a\in A(s)} \sum_{s'\in\mathcal{S}} P(s'|s,a)\big[C_a(s,s')+V^\pi(s')\big]\Big\} \qquad (14)$$

They show that MDPs with dead ends can be solved with VI that uses Eq. (14) for updates. Moreover, all heuristic search algorithms for solving SSPs (such as RTDP) and their guarantees apply to this type of MDPs if they use Eq. (14) in lieu of Bellman's update.

Real-Time Dynamic Programming

RTDP (Barto, Bradtke, and Singh 1995) is a heuristic-search DP algorithm for solving non-deterministic planning problems with full observability. In relation to other dynamic programming methods, RTDP has two benefits. First, it is focused, namely it updates only states that are encountered in the search and thus relevant to the problem solving. Second, it has a good anytime behavior, i.e., it produces good policies fast and these policies improve smoothly with time.

RTDP works by repeated trials or runs (see algorithm 1). Each trial starts at the initial state s_0 and ends in a goal state or a dead-end state. At each step, action selection is greedy based on the current value function, and outcome selection is stochastic according to the distribution of possible successor states given the chosen action. The values $V(s)$ of the visited states are updated along the way, using Bellman's equation (Eq. (5)). The initial values of $V(s)$ are given by an heuristic function $h(s)$.

After the termination condition is met, a coverage path is built by following the greedy policy from the starting state to a goal state. In this final phase, we never enter a dead-end state; whenever the robot visits a threat point, the outcome that represents its survival of the threat is chosen deterministically. This way we guarantee that a complete coverage path is created, i.e., a path that covers all the cells in the target area. In contrast, RTDP trials may be terminated before the entire area is covered. This helps the algorithm focus on updating states which the robot has more chance to reach.

From (Barto, Bradtke, and Singh 1995), it is known that under conditions S1-S8 for SSPs, if the initial value function is admissible, i.e., $h(s) \leq V^*(s)$ for every state s, then repeated RTDP trials eventually yield optimal values $V(s) = V^*(s)$ over all relevant states (states that can be reached by at least one optimal policy). In our experiments we have used the admissible heuristic function $h \equiv 0$.

Frontier-Based RTDP

A good heuristic can lead to faster convergence of RTDP. However, choosing a good admissible heuristic function is often a non-trivial task. Moreover, in a huge state space such as we have here, finding a good heuristic function may not be enough. Initial results from our empirical evaluation have indicated that one of the main reasons for the slow convergence of RTDP is that its trials a waste considerable

Algorithm 1 Real_Time_Dynamic_Programming

Input: a grid G, a starting cell c_0, a termination criterion ϵ
Output: a coverage path P that covers all reachable cells in G from c_0

function RTDP(s_0) // *s_0 is the initial state*
  while $\max_{s \in visited}$ RESIDUAL(s) $> \epsilon$ **do**
    RTDPTRIAL(s_0)
  return BUILDCOVERAGEPATH(s_0)

function RTDPTRIAL(s) // *Execute one trial of RTDP*
  while not GOAL(s) and $s \neq s_d$ **do**
    // *Pick best action and update hash*
    $a \leftarrow$ GREEDYACTION(s)
    UPDATE(s, a)
    // *Stochastically simulate next state*
    $s \leftarrow$ CHOOSENEXTSTATE(a)

function GOAL(s)
  return if all reachable cells from c_0 are covered in s

function INITSTATE(s) // *Implicitly called the first time each state s is touched*
  $s.V \leftarrow h(s)$

function GREEDYACTION(s)
  return $\arg\min_{a \in A(s)}$ QVALUE(s, a)

function QVALUE(s, a)
  return $\sum_{s' \in \mathcal{S}} P(s'|s,a)\big[C_a(s,s') + s'.V\big]$

function UPDATE(s, a)
  $s.V \leftarrow$ QVALUE(s, a)

function CHOOSENEXTSTATE(s, a)
  Choose s' with probability $P(s'|s,a)$
  return s'

function RESIDUAL(s)
  $a \leftarrow$ GREEDYACTION(s)
  return $|s.V -$ QVALUE(s, a)$|$

function BUILDCOVERAGEPATH(s)
  Create a new coverage path P
  Add starting cell c_0 to P
  while not GOAL(s) **do**
    $a \leftarrow$ GREEDYACTION(s)
    Make the robot move according to action a
    Add the cell c where the robot is located to P
    // *Deterministically simulate next state*
    $s \leftarrow s$ with cell c marked as visited and robot's location is at c
  return P

amount of time moving back and forth between already visited states. For example, let us examine states s_1 and s_2 from the search subgraph depicted in Figure 1. Since both cells $(1,1)$ and $(1,2)$ are safe, an RTDP trial would travel back and forth between states s_1 and s_2 until the estimated cost of the repeated transition between them becomes higher than the cost of moving to one of the dangerous cells $(2,1)$ or $(2,2)$. Clearly, these repeated transitions cannot be part of the trajectory followed by the optimal policy (they only increase the cost of the path to the goal), and thus should be eliminated from the search.

Frontier-Based RTDP (algorithm 2) avoids such fruitless cyclic returns in the search graph, by maintaining a list of *frontier* states, defined as states that separate the covered regions of the search space from those uncovered. Each time a new state is encountered by an RTDP trial, it goes over all its possible successors, and adds to the frontier list all the unvisited successors that are not already in this list. A state is taken out of the frontier list once it is visited by the trial.

At each step of the trial, FBRTDP examines all the possible paths from the current state to one of the frontier states, and chooses the path with the minimum expected cost according to the current value function. To allow a transition from any given state to a frontier state, we extend the set of actions A in the MDP model with the following definition.

Definition 1. ***Composite action*** *$\hat{a}$ is an action that consists of a sequence of actions $(a_1, ..., a_n)$ from A.*

The possible outcomes of a composite action consist of all the states that could be reached by an RTDP trial following the sequence $(a_1, ..., a_n)$.

In the adversarial coverage case, any composite action has only two possible outcomes: reaching the destination cell of the final action in the sequence $(a_1, ..., a_n)$ or entering the dead state. The probabilities of these outcomes depend on the threat probabilities of the cells $(c_1, ..., c_n)$ encountered along the path taken by the robot following the actions in $(a_1, ..., a_n)$. More specifically, the probability of the first outcome, in which the robot is able to visit all the cells $(c_1, ..., c_n)$ without being hit by a threat, is:

$$P(s'|s,\hat{a}) = \prod_{i=1}^{n}(1 - p_i) \tag{15}$$

whereas the probability of the second outcome, in which the robot is stopped by a threat along the path, is complementary to the probability of the first outcome, i.e.,

$$P(s_d|s,\hat{a}) = 1 - P(s'|s,\hat{a}) \tag{16}$$

More generally, to compute the probability $P(s'|s,\hat{a})$ of each outcome of a composite action $\hat{a}$, we need to add up the probabilities of all the possible paths from the current state s to the destination state s' of that outcome.

We now define the cost of a composite action $C_{\hat{a}}(s,s')$ as the sum of the costs of all its primitive actions $(a_1, ..., a_n)$. If we denote by $(s_0, ..., s_n)$ the set of states visited by the RTDP trial following the actions $(a_1, ..., a_n)$, starting from the current state s_0, then the cost of $\hat{a}$ is:

$$C_{\hat{a}}(s_0, s_n) = \sum_{i=1}^{n} C_{a_i}(s_{i-1}, s_i) \tag{17}$$

By linearity of expectation, the expected cost of a composite action $\hat{a}$ is the sum of the expected costs of all its primitive actions, i.e.,

$$E\big[C_{\hat{a}}(s_0, s_n)\big] = \sum_{i=1}^{n} E\big[C_{a_i}(s_{i-1}, s_i)\big] \tag{18}$$

In order to find a path with minimal expected cost from the current state to a frontier state, at each step of the trial we

build a subgraph of the search space that consists of the visited states so far and the frontier states. Then, we execute Dijkstra's shortest paths algorithm on this subgraph, where the weight w_{ij} of the edge connecting states s_i and s_j is defined as the expected cost of the action leading from state s_i to s_j, i.e., $w_{ij} = E\big[C_a(s_i, s_j)\big]$.

Dijkstra's algorithm finds paths with minimum expected costs between the current state s_0 and all the frontier states in this subgraph. The next action chosen by FBRTDP is the composite action that leads from the current state s_0 to the frontier state with minimum expected cost path from s_0.

Additionally, one can exploit domain-specific knowledge to narrow down the set of frontier states and thus prune more irrelevant states from the search. Specifically, in the adversarial coverage case, we consider only states in which the robot reaches an unvisited cell in the map as frontier states. For instance, let us examine state s_2 from Figure 1. Although this state has not been encountered in the search before, we can treat it as a non-frontier state, since in this state the robot returns to an already visited cell $(1, 1)$. Thus, the possible successor frontier states of s_1 that should be considered are the states in which the robot reaches one of the unexplored cells $(2, 1)$ or $(2, 2)$.

We now prove that FBRTDP has the same optimal convergence guarantees as RTDP.

Theorem 2. *Under conditions S1-S8, if the initial value function is admissible, repeated FBRTDP trials eventually yield optimal values $V(s) = V^*(s)$ along every optimal path from the initial state to a goal state.*

Proof. The main idea of the proof is to show that it is enough to consider the paths to frontier states from any given state in order to reach the optimal value function.

The first observation is that FBRTDP preserves the non-overestimating property of h when visiting a state and updating its value. Let us denote by $F(s)$ the set of frontier states that can be reached from a given state s. Assuming that the h values of $F(s)$ do not overestimate the expected cost to reach the goal, then after adding paths with minimum expected cost from s to each of these frontiers, the minimum of the resulting values cannot overestimate the expected cost to the goal from the given state.

We now define the value $V(s)$ of a state s to be *consistent* with the frontier states that can be reached from it, if $V(s) = \min_{s' \in F(s)} \big[E[C(s, s')] + V(s')\big]$, where $E[C(s, s')]$ is the expected cost of the optimal path from s to s'.

Now, assume the converse of the theorem, that after an infinite number of trials, there exists a state along an optimal path from the initial state to a goal whose value is not optimal. Assuming that h of all goal states is zero, if the value of any state along any path to a goal state is not optimal, then some frontier state along the same path must be inconsistent. This follows formally by induction on the distance from the goal.

If there exists a frontier state whose value is inconsistent, then there must exist at least one such state in an arbitrary ordering of the states. Call such a state x. By assumption, x lies along an optimal path from the initial state s to a goal state. In addition, since all the h values are non-overestimating and this property is preserved by FBRTDP, the values of all the states along the optimal path from s to x, are less than or equal to their optimal values. This ensures that state x will eventually be visited by FBRTDP. When it is, its value will become consistent with the frontier states that can be reached from it, thus violating the assumption that it is the least inconsistent frontier state in some ordering. Therefore, the value of every state along an optimal path from the initial state to a goal state must eventually reach its optimal value. □

Algorithm 2 Frontier_Based_RTDP

Data structures: $frontier$ - set of frontier states
$visited$ - set of states already visited by the current trial

function FBRTDP(s_0) // *s_0 is the initial state*
  while $\max_{s \in visited}$ RESIDUAL(s) $> \epsilon$ **do**
    $visited \leftarrow \{s_0\}$
    $frontier \leftarrow$ all successors of s_0
    FBRTDPTRIAL(s_0)

function FBRTDPTRIAL(s) // *Execute one trial*
  while not GOAL(s) and $s \neq s_d$ **do**
    // *Pick best composite action and update hash*
    $\hat{a} \leftarrow$ GREEDYCOMPOSITEACTION(s)
    UPDATE($s, \hat{a}$)
    // *Stochastically simulate next state*
    $s \leftarrow$ CHOOSENEXTSTATE($\hat{a}$)
    if $s \notin visited$ **then**
      Add s to $visited$
      UPDATEFRONTIER(s)

function GREEDYCOMPOSITEACTION(s)
  Build a graph G that consists of the states in $visited \cup frontier$ and its edge weights defined as the expected costs of the state transitions
  Run Dijkstra on the graph G starting from s
  Find a frontier f with minimum cost path from s
  Let $\hat{a} = (a_1, ..., a_n)$ be the sequence of actions leading from s to f on the minimum cost path
  return $\hat{a}$

function QVALUE($s, \hat{a}$)
  return $\sum_{s' \in S} P(s'|s, \hat{a})\big[C_{\hat{a}}(s, s') + s'.V\big]$

function UPDATE($s, \hat{a}$)
  $s.V \leftarrow$ QVALUE($s, \hat{a}$)

function CHOOSENEXTSTATE($s, \hat{a}$)
  Choose s' with probability $P(s'|s, \hat{a})$
  return s'

function UPDATEFRONTIER(s)
  Remove s from $frontier$
  for every successor state s' of s **do**
    if $s' \notin visited$ and $s' \notin frontier$ **then**
      Add s' to $frontier$

Empirical Evaluation

In this section we evaluate FBRTDP in relation to RTDP and three other algorithms: VI, the standard dynamic programming algorithm, LRTDP (Labeled RTDP) (Bonet and Geffner 2003), and GAC (Greedy Adversarial Coverage),

the state-of-the-art solution to the adversarial coverage problem as described in (Yehoshua, Agmon, and Kaminka 2014). We use a specific map to illustrate the operation of the algorithms and we also report on the statistical analysis of their behavior based on multiple randomly generated maps with varying parameters.

Figure 2 shows an example for the optimal safest coverage path found by VI on a map of size 7×7. The map contains 25% threat points with 5 different threat probabilities between 0.006 and 0.03, and 30% obstacles. Obstacles are represented by black cells, safe cells are colored white and dangerous cells are represented by 5 different shades of purple. Darker shades represent higher values of p_i (more dangerous areas). The number of visits to each cell along the coverage path is indicated within that cell. The termination criterion was set to $\epsilon = 0.1$.

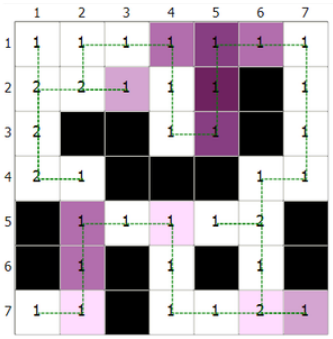


Figure 2: An optimal coverage path generated from a Value Iteration run.

As can be seen, the coverage path revisits only a single threat point, which has the lowest threat probability. The other five revisits are to safe cells. This coverage path is optimal, since any coverage path of this map must revisit at least one threat point (there is a threat point located next to the two lower corners and the robot must get in and out of at least one of these corners in order to complete the coverage). The probability to complete the coverage path generated by VI was 49.5%, and its total length was 40. Running GAC on the same map generated a coverage path with 44.71% probability to complete (containing 4 revisits to threat points) and total length of 72. RTDP, LRTDP and FBRTDP converged to the same optimal solution as VI on this map, albeit in a much shorter time. The curves in Figure 3 display the evolution of the expected cost to the goal as a function of time for the different algorithms. FBRTDP shows the best profile, converging to the optimal policy in only 0.429 seconds, while RTDP, LRTDP and VI converge to the optimal policy in 541, 530, and 803 seconds, respectively.

For map sizes larger than 7×7, the MDP's state space exceeded the available memory, thus VI could not be executed for such maps. Moreover, in such maps, RTDP's and LRTDP's convergence was very slow (it took a few hours for them to converge on maps of size 8×8).

Therefore, for large-sized maps we have analyzed the performance of only FBRTDP and GAC. Figure 4 shows the

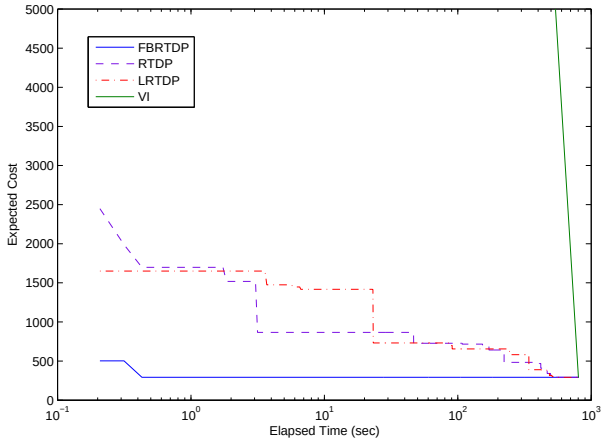


Figure 3: Expected cost to the goal vs. time for VI, RTDP, LRTDP and FBRTDP. The time axis is plotted on a logarithmic scale.

completion probability and the path length obtained by both algorithms for varying α/β ratios between 0.001 and 1000. The results are averaged on 30 random maps. In all experiments we have used map sizes of 20×20, the ratio of obstacles was 30%, the ratio of threats was 30% and the number of threat levels was 5. The locations of the threat points and the obstacles were randomly chosen. Note that for maps of this size, FBRTDP did not converge (i.e., the residual didn't get below ϵ) in a reasonable amount of time, thus we halted it after 1000 trials.

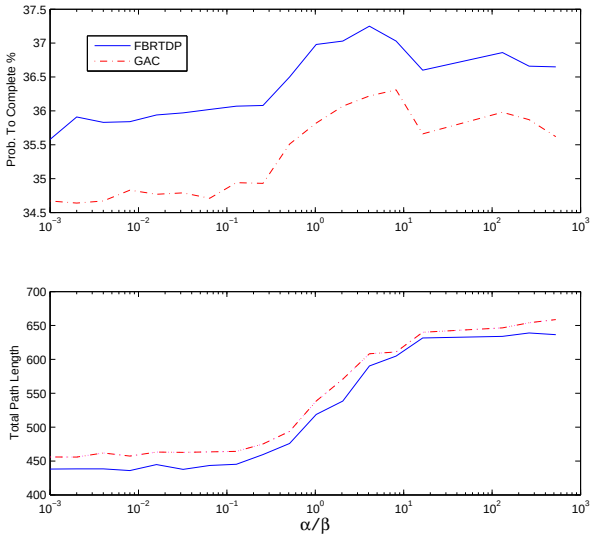


Figure 4: Probability to complete the coverage and total path length for different risk and time levels. x axis is plotted on a logarithmic scale.

As can be seen, the coverage path length increases as the

risk factor α become more dominant in both algorithms. The probability to complete increases until the ratio α/β is around 5 and then it starts to decrease. This is due to the fact that when α is too high, the algorithms try to avoid visiting higher-level threats as much as possible, which makes them revisit lower-level threats more times, thus the coverage path gets longer and riskier.

In all experiments, FBRTDP consistently outperforms the greedy algorithm in terms of both the completion probability and the path length. On average, FBRTDP achieves about 1% increase in the robot's survivability and 5% decrease in the path length compared to GAC (which is statistically significant; one-tailed t-test $p = 7.43 \cdot 10^{-16}$). As can be seen from the graph, changing the ratio between the risk and the time factors (α/β) from 0 to ∞, under the given map settings, can change the survivability probability by only 1.5% in both algorithms. Thus, a 1% increase in the robot's survivability is quite dramatic. The absolute difference between the algorithms' results depends upon map settings. For example, when the threats ratio was decreased to 25%, FBRTDP attained a robot's survivability probability which was 3% higher than GAC.

On the down-side, FBRTDP's average running time was significantly higher than GAC's (179 seconds in FBRTDP, 0.154 seconds in GAC). This difference is caused by the high number of FBRTDP trials that was needed in order to reach convergence. However, FBRTDP typically outperforms GAC after a small number of trials (an FBRTDP trial follows a greedy policy which resembles the greedy behavior of GAC).

Conclusions and Future Work

We have described how to model the robotic adversarial coverage problem as an MDP. We have shown how the model can be used to find an optimal solution to the problem on small-sized maps, and obtain significant improvement over the state-of-the-art solution for larger maps. To the best of our knowledge, this is the first time that MDPs have been used to represent problems in the robotic coverage field.

We have also introduced FBRTDP, a new improvement to RTDP, which maintains a list of frontier states and extends the set of actions that can be used by the model. FBRTDP provides significant speedup, allows RTDP to solve the adversarial coverage problem on much larger maps, and has the same optimal convergence guarantees as RTDP.

In the future we plan to use the MDP model to handle other variants of the adversarial coverage problem, such as a variant in which threats may cause time delays instead of completely stopping the robot. We also intend to evaluate FBRTDP on other planning problems and compare its performance to other heuristic algorithms for solving MDPs.

References

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72(1):81–138.

Bellman, R. 1957. *Dynamic programming*. Princeton University Press.

Bertsekas, D. P. 1995. *Dynamic programming and optimal control*, volume 1 and 2. Athena Scientific.

Bonet, B., and Geffner, H. 2003. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. of ICAPS*, volume 3, 12–21.

Colegrave, J., and Branch, A. 1994. A case study of autonomous household vacuum cleaner. *AIAA/NASA CIRFFSS* 107.

Gabriely, Y., and Rimon, E. 2003. Competitive on-line coverage of grid environments by a mobile robot. *Computational Geometry* 24(3):197–224.

Galceran, E., and Carreras, M. 2013. A survey on coverage path planning for robotics. *Robotics and Autonomous Systems* 61(12):1258–1276.

Kolobov, A.; Mausam; and Weld, D. 2012. A theory of goal-oriented mdps with dead ends. In *Proc. of the Conference on Uncertainty in Artificial Intelligence (UAI-12)*, 438–447.

Little, I., and Thiebaux, S. 2007. Probabilistic planning vs. replanning. In *ICAPS Workshop on IPC: Past, Present and Future*.

Luo, C.; Yang, S. X.; Stacey, D. A.; and Jofriet, J. C. 2002. A solution to vicinity problem of obstacles in complete coverage path planning. In *Proc. IEEE International Conference on Robotics and Automation (ICRA-02)*, volume 1, 612–617.

Nicoud, J. D., and Habib, M. K. 1995. The pemex-b autonomous demining robot: perception and navigation strategies. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems, 'Human Robot Interaction and Cooperative Robots'*, volume 1, 419–424.

Nikolova, E., and Karger, D. R. 2008. Route planning under uncertainty: The canadian traveller problem. In *Proc. of the Twenty-Third Conference on Artificial Intelligence (AAAI-08)*, 969–974.

Nishizeki, T.; Asano, T.; and Watanabe, T. 1983. An approximation algorithm for the hamiltonian walk problem on maximal planar graphs. *Discrete applied mathematics* 5(2):211–222.

Papadimitriou, C. H., and Yannakakis, M. 1989. Shortest paths without a map. In *Automata, Languages and Programming*. Springer. 610–620.

Yehoshua, R.; Agmon, N.; and Kaminka, G. A. 2013. Robotic adversarial coverage: Introduction and preliminary results. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-13)*, 6000–6005.

Yehoshua, R.; Agmon, N.; and Kaminka, G. A. 2014. Safest path adversarial coverage. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-14)*, 3027–3032.

A Framework for Performance Assessment of Autonomous Robotic Controllers

Pablo Muñoz [1] and **Amedeo Cesta** [2] and **Andrea Orlandini** [2] and **María D. R-Moreno** [1]
[1] Department of Automatics, University of Alcalá, Alcalá de Henares, Spain
{pmunoz, mdolores}@aut.uah.es
[2] Institute of Cognitive Science and Technology, CNR, Rome, Italy
{amedeo.cesta, andrea.orlandini}@istc.cnr.it

Abstract

This work describes a framework to assess the performance of autonomous software for robotics platforms. The motivation for such an effort is twofold: (a) the difficulty to generate intensive test campaign for a given robotic architecture; (b) the lack of a general approach to assess and compare different planning and execution approaches for the same robotic platform. The produced framework, called OGATE, supports the integration, testing and operationalization of an autonomous robotic controller. It allows to run series of plan execution experiments while collecting and analyzing relevant parameters of the system under a unified and controlled environment. The first part of the paper presents the framework, an evaluation methodology for autonomous controllers and an initial deployment of OGATE to support experiments using a specific robotic system as a case study. The second part of the paper summarizes the lessons learned exploiting the proposed methodology and OGATE as an automated testbench to analyze different planning policies for the deliberative component of the targeted robot.

Introduction

The interleaving of planning and execution is a key reference problem for the Planning and Robotics research communities. For example it is one of the main issues to be addressed when deploying autonomous control systems in simulated and/or real robotics platforms – see e.g., (Gat 1992; Alami et al. 1998; Aschwanden et al. 2006; Nesnas et al. 2006; Py, Rajan, and McGann 2010). As a common practice, such research efforts rely on rather specific validation methodologies, experimental settings and assessment analysis in a manner that leads them to be hardly exportable and reproducible on different architectures. As a consequence, the process of generating reports after empirical evaluation with robots featuring plan-based autonomous capability often lacks a comparable testing methodology. Specifically, the presence of uncertainty, errors and robot decision capabilities (based on how the world is perceived) allows to provide a sort of *"proof of concept"* affected by (i) usage of subjective and/or insufficiently general performance metrics, (ii) difficulties in reproducing results and (iii) heterogeneity of experimental conditions (Fontana, Matteucci, and Sorrenti 2014). An interesting open issue consists of defining an evaluation methodology for control systems capable of being exportable and reproducible with different plan-based controllers for autonomous robotics.

Paper Contribution. This paper aims at contributing according to the following directions: (1) a methodology for evaluating autonomous controllers is defined and discussed; (2) such a methodology is operationalised in a case study constituted by a specific goal-based autonomous controller of a rover in a space exploration domain; (3) the performance of the autonomous controller is assessed considering a set of metrics whose measurements are collected during an automatically generated test campaign. The proposed framework (Muñoz et al. 2014) aims at becoming a general and domain independent tool that, following a well structured methodology, can support the analysis and comparison of autonomous controllers performance, based on objective metrics and well suited experimental campaigns. In this paper we only present the results obtained using a TimeLine Based autonomous controller but we are also working on a PDDL-based one. In the future, we will be able to show the comparison between them. The framework allows to define metrics according to specific evaluation goals, to define a set of application scenarios to be exploited in order to evaluate actual robotic platforms or associated simulators under controlled and reproducible experimental conditions. The tool is in charge of supervising and monitoring the controller execution by inspecting internal monitors of the different components and retrieving relevant information about its performance. Finally, the collected information are exploited to generate detailed reports to support assessments based on the analysis of the considered metrics.

Related Works. Evaluating and characterizing autonomous controllers have been investigated in different perspectives. On the one hand, there are theoretical works that aim to define the relevant parameters to measure for an autonomous system (Ad Hoc ALFUS Working Group 2007; Huang et al. 2010) or those who try to create valid methodologies for the testing process (Hudson and Reeker 2007; Gertman et al. 2007). On the other hand, the are the robotics competitions which allow us to compare different solutions for the same problem with

different platforms/controllers (del Pobil 2006; Behnke 2006). Notwithstanding the relevance of such works, they are focused on non-objective and weak evaluation criteria (e.g., (Orebäck and Christensen 2003; McWilliams et al. 2007)), while others rely on expensive robotic platforms that are not accessible to the wide research community. In any case, the complexity of exploiting these systems in an automated test campaign remains an open issue.

Plan of the Paper. In the next section we present the OGATE tool able to perform automatic campaigns to characterize and evaluate autonomous controllers. Then, an evaluation methodology is presented. Next, we present the GOAC controller and the space exploration domain that are used to demonstrate the OGATE capabilities to perform an automated test-bench to analyze the impact of different planning policies of the deliberative component of the architecture. After, we discuss the performance evaluation of the GOAC controller as a function of the deliberative component for different space exploration scenarios. Some conclusions end the paper.

The OGATE Tool

Plan-based deliberative systems are usually the top layer of complex autonomous controllers specifically designed to control a specific robotics platform to perform determinate duties. Hand tailored testing and verification tasks are often exploited to validate such deliberative systems. More in general, subparts of the control architectures can be evaluated in a stand-alone manner via particular test-bed. But testing robustness, adequacy and performance for the whole architecture is not a trivial task as it requires to collect and to analyse relevant data from all the parts of the control system while the test bed should be properly designed and not just covering more than just typical scenarios. The authors current work aims at designing and developing a software framework called OGATE (in (Muñoz et al. 2014) an initial report can be found) whose main objective is to support the deployment of control architectures for robotics platforms and to perform sets of reproducible experiments.

OGATE offers capabilities for instantiating and activating the required components of an autonomous controller within a scenario defined by a user, supervises the plan execution and generates a report with information gathered during the test. Furthermore, it constitutes also an interactive tool to help designers and operators of autonomous controllers providing a unified interface for in-execution control and inspection of the controlled system. In this regard, OGATE aims at providing an environment to test the features of goal oriented controllers as well as to generate reports based on quantitative performance analysis by means of software internal monitors of the controlled system after experiments. A basic concept in OGATE is the one of *mission* that identifies a combination of goals, plans and robot configuration details over a defined interval of time. The infrastructure of the system relies on three main modules (see Figure 1) to support the test bed lifecycle for autonomous agents, namely: mission specification, execution and report.

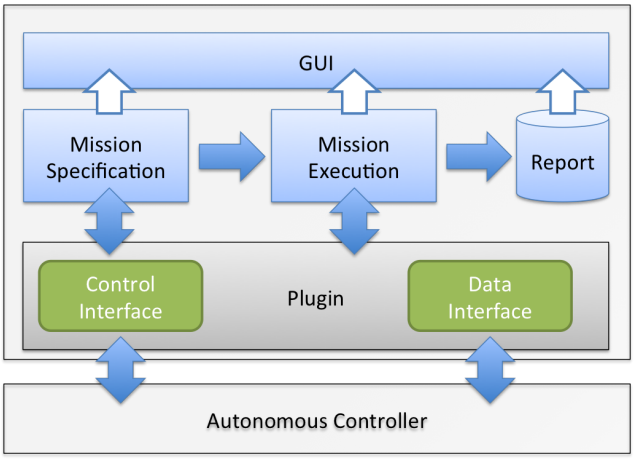


Figure 1: The OGATE framework

Mission specification. In order to define functionalities and required goals, it is first required to specify the mission *configuration* for the autonomous controller, i.e., the components of the control architecture and the platform over which they operate. In this regard, OGATE provides a suitable interface to configure the components of the controlled system, such as the deliberative module. In general, the mission specification includes the configuration of the different components (executive, deliberative, etc.) involved in the mission. Then, a suitable support is provided also to support the definition of the experimental scenario, i.e., domain and a problem specification typically required by these AI controllers. Namely, the definition of causal and temporal interactions between different elements of the system and, initial facts and desired objectives of the mission.

Mission execution. OGATE supports also experiments execution providing a framework to deal with the complexity of the underlying architecture and the different subcomponents, to execute the user defined missions and to gather relevant data in order to provide the user with a report about general controller performance. Also, the user is able to interact with the controlled system during execution, modifying internal parameters or including new mission goals to change the nominal execution so as to test the system also under dynamic conditions, e.g., focusing on the replanning capabilities of the deliberative component. Then, using specific operational interfaces, OGATE is able to control and gather data related to execution accessing the single components of the autonomous controller. It is worth observing that the system has the capability for both single run or batch experiment set run hence facilitating data gathering.

Report. At the end of the execution of an experiment, OGATE provides the user with a report about the collected data. In this regard, OGATE provides a human-readable report containing detailed information about the autonomous controller performance while facing the mission. The system is then open to be enriched with user-defined metrics for assessing the performance of deliberative components enabling the evaluation of the considered autonomous

controller.

In order to control the autonomous controller and access information, some parts of the control architecture must be accessible. In this regard, OGATE defines a small set of simple interfaces allowing the control of different aspects of a component (see again Figure 1). An OGATE plug-in is a component of the system that implements some functionalities accessible through such interfaces, giving access to OGATE to control its execution and to retrieve data from it. The interfaces are: a **Control interface** to provide the basic functionality to safely run, pause or stop the autonomous controller and its submodules also monitoring the status of the components and allowing the detection of non-nominal states; a **Data interface** to supply a bidirectional channel to retrieve the relevant data and to modify the internal status of the components such as, for instance, including new goals during the execution of deliberative components.

Also, the specification of a test-bench over that mission configuration consists of evaluating different control architectures or various configurations for a control architecture to select the best one for a particular mission, or the evaluation of the performance for a particular control architecture over a set of missions, to evaluate and improve the components employed. The OGATE system will be able to support these tests in an automated way.

Then, in order to work with OGATE, the user shall provide a set of elements: (a) the configuration of the different components of the autonomous controller. This is done using an ad-hoc XML configuration file containing the required instructions to instantiate the particular autonomous controller; (b) a scenario description over which the autonomous controller is to be tested. Defining a pair of domain and problem models, and other configurations files to define the behaviors of the different components; (c) a metrics definition file in which the user can set the different metrics to be considered. Each metric is defined by its name, the value range (lower and upper bounds) and the importance of the metric in the final evaluation (weight).

Given the above information, OGATE is able to instantiate the autonomous controller exploiting the information contained in the configuration file. Then, when all the considered components are properly set, it is possible to start their execution and monitor the status of the components collecting the information related to the defined metrics. As soon as the execution ends, OGATE generates a report including minimum, maximum and average value for each metric, and a graphical report with the complete evaluation of the controller performance.

Given this software infrastructure we now present an approach to evaluating plan-based autonomous controllers and then we will describe the approach at work of a deliberative control architecture, commenting the results that are triggered by the use of the system.

Evaluation methodology

In general terms, given an autonomous controller to be assessed, a set of evaluation objectives should be isolated and some specific performance metrics should be identified and defined accordingly. Then, a set of suitable tests should be defined and performed so as to collect relevant information constituting a quantitative basis for the evaluation process. Finally, reports should be generated to point out measurements indicating the performance of the controller according to the evaluation objectives and metrics defined in the first phase.

More in detail, the methodology proposed to analyze and evaluate autonomous controllers can be described as the composition of three phases: evaluation design, tests execution and, report and assessment.

Evaluation Design. First, the **identification of evaluation objectives** is to identify which is the evaluation target. In fact, according to the evaluation target different aspects may result relevant (or not). For instance, measuring the deliberation time or considering the number of dispatched goals in different scenarios could provide relevant information about the behaviour of the autonomous controller. In this case, very specific parameters can be considered and analysed. More in general, a set of parameters applicable to any deliberative system can be considered in order to enable also the possibility to compare performance of different control systems in the same operative scenario.

According to evaluation objectives, a **metrics definition** task is to define parameters that should be measured during execution. This is key as the result of the evaluation strongly depends on the selected metrics. It is important to define (at least) a small set of metrics that can be commonly applicable to autonomous controllers, such as the time spent deliberating and the total time to accomplish all goals. Anyway, focusing on a particular autonomous controller allows to consider specific metrics for that controller during tests.

Then, the **definition of different scenarios and configurations** to be tested should be implemented. The scenarios can be defined as the set of constraints and goals that the autonomous controller takes as input. In general, a representation of the interactions between the robotic platform and the external world coupled with a given set of goals to be accomplished. However, to also deal with uncertainty, scenarios should be defined considering external agents that can dynamically send additional goals or some failures that can occur during execution. Such scenarios definition requires advanced capabilities such as replanning and failure recovering schemes. More than one scenario can be defined in order to investigate the behaviour of the autonomous controller under different conditions.

Tests Execution. Performing tests entails the execution of each scenario that is to be monitored. In case, uncertain and/or uncontrollable tasks are part of the problem, each scenario should be performed several times, collect average behaviours and define metrics values. In this regard, a **scenario instantiation** step is required to generate the set of configuration files combining the sets of planning domains and required goals. Also, autonomous controllers can be deployed with different internal settings, so, scenarios instantiation should consider also to enable the execution of tests

under different conditions. Then, actual **tests execution** is needed. This is an important step for instantiating, executing, monitoring and collecting the data of several executions of an autonomous controller in a given scenario.

Report and Assessment. Once all the tests are completed, a **Report** on the information gathered during the several executions. Reports are to provide an insight of the controller behaviours providing values for each metric as well as generating synthesised views, e.g., by means of graphical representations to support the users while analysing system performances. In fact, the information provided within reports is to inform users and enable a **performance assessment** enabling an objective evaluation of the control architecture in the different scenarios. After execution, a huge amount of generated data is expected and then a general representation for the data produced is to be defined.

Metrics Definition and Presentation

Definition and presentation of metrics deserve a more detailed discussion and, in the following, a detailed formalization is provided. In the above methodology, a set of metrics M is considered, being a metric denoted as $\mu_i \in M$ and defined in a range $\mu_i^{lb} \geq \mu_i \geq \mu_i^{ub}$ with μ_i^{lb} and μ_i^{ub} are (respectively) the lower and upper bounds for the i metric. Also, for each metric an extra parameter is to be considered, i.e., the weight, μ_i^W, that represents the *relevance* of the metric within the global evaluation. Considering the size of M (i.e., the number of defined metrics) as n, the sum of all weights is supposed to be 100:

$$\sum_{i=0}^{n} \mu_i^W = 100 \tag{1}$$

After execution, the average value for each metric, μ_i^V, is considered in the report and this value is considered to compute a *metric score* μ_i^S as follows:

$$\mu_i^S = \left[100 - \left(\frac{100}{|\mu_i^{ub} - \mu_i^{lb}|} \cdot \mu_i^V\right)\right] \cdot \frac{\varepsilon^C}{\varepsilon^T} \tag{2}$$

considering that the upper bound of the metric is the worst score. If the metric value is out of the defined range, its score is 0. Last factor, $\varepsilon^C/\varepsilon^T$, expresses the impact of execution failures in the metrics scores, being ε^C the number of correct executions and ε^T the total runs.

As stated before, a suitable way to provide reports is by means of graphical representations. For example, in ALFUS (McWilliams et al. 2007) and PerMFUS (Huang et al. 2010), a three axis representation based on the mission and environment complexity and human independence is presented. In a similar way, here, a circular graphic representation, such as the one depicted in Figure 2, is proposed to represent the autonomous controller performance. Such representation presents three different areas. Namely, starting from the center, the Global Score (GS), the execution times and the metrics scores area.

The Global Score (GS) presented in the center of the figure represents a synthetic evaluation for the architecture in a scale between 0 and 10, that can be compared with the Sheridan's model (Parasuraman, Sheridan, and Wickens 2000). In that model, the score increases with the level of autonomy demonstrated by the controller, being 10 a fully autonomous system. In our evaluation, a higher score represents a better evaluation as a function of the defined metrics. To compute the GS value, only metrics scores are considered (execution times are not considered), being the score directly proportional to the filled area of the ring, and computed as follows:

$$GS = \frac{\sum_{i=0}^{n} \left(\mu_i^S \cdot \mu_i^W\right)}{1000} \tag{3}$$

Surroundings the GS area, three circular bars are depicted. These bars represent the average time required by the considered autonomous controller to complete each scenario. Starting from the center, these bars represent: the execution time in (i) nominal conditions, (ii) in dynamic goal injection conditions and (iii) the presence of execution failures.

Finally, the external ring in the chart is decomposed into four quadrants. The smaller circumference of the ring represents the smaller score for a metric ($\mu_i^S = 0$, when the metric score is equal or bigger to its upper bound), while the outside circumference is the best value ($\mu_i^S = 100$, or a metric value closer to the lower bound). In each quadrant there are one or more metric scores represented as a filled circular sector. So, depicting a metric requires metric weight (μ_i^W provided by the user) and metric score (μ_i^S obtained from the execution using eq. 2). As a result, the higher is the weight of the metric and its score, the higher is the filled area of the ring. Then, a evaluation with a GS of 10 is this one in which the metrics score fills all the ring.

The metrics area is decomposed in four quadrant to allow a clustering of the metrics. The clustering criteria is not fixed and is up to the user to define it. A good practice may be to organize the metrics in the four quadrants grouping them by, for example, the degree of autonomy that the metrics represent. In that case, this implies to follow a schema in which metrics related to the functional support are clustered in the

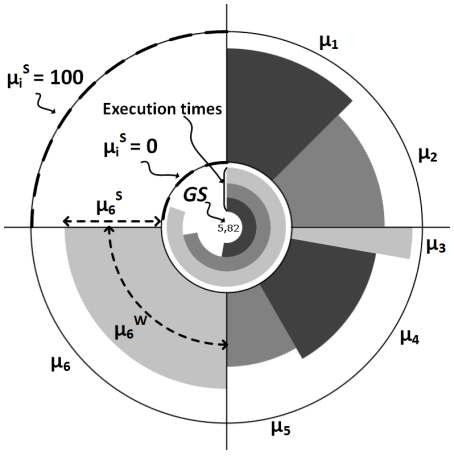


Figure 2: The OGATE graphical report.

first quadrant while those related to the deliberative system are presented in the fourth quadrant. Besides, while focusing on a small set of metrics, the quadrants can be exploited to present the same metrics applied to four different instances of the same scenarios. In general, the representation may be customised according to the objectives of the evaluation.

For example, in the graph presented in Figure 2, the values for lower and upper bounds and the weights (μ_i^W) are provided by the user. The value of the metrics bounds can be obtained through empirical evaluations or predictions of the controller performance. The unit of the metrics are dependent of the value measured, they can be seconds, a percentage or another unit. The weight of each metric shall be decided by the user, by identifying the relative importance of each metric for the evaluation process, ensuring that complies with the eq. 1. The metric value (μ_i^V) is obtained during the tests execution and the score (μ_i^S) is computed at the end. For the metric score the units have not sense, the scores are normalized in a range between 0 and 100. With these values, the global score for that architecture tested over a particular scenario and with this set of metrics is 5.82 as depicted in the center of the fig. 2, computed through the eq. 3. The values for the execution time are only presented as an example of how a complete evaluation looks.

This methodology constitutes a generic and reproducible process to evaluate autonomous controllers while considering varying execution scenarios. In this regard, the definition of metrics, i.e., weights, bounds and experimental cases, is the basic step on which rely to reproduce the evaluation results. Also, the proposed graphical representation provides a synthetic and standard approach to summarize performance results in test campaign, which allow users to analyze and compare different aspects of controllers performance in an straightforward manner.

Case study: GOAC in planetary exploration

Thanks to ESA (European Space Agency) Robotic Department we have been able to use the GOAC autonomous controller (Goal Oriented Autonomous Controller (Ceballos et al. 2011)) an effort from the agency to create a reference platform for robotic software for different space missions. The GOAC architecture is the integration of several components: (a) a timeline-based deliberative layer which integrates a planner, called OMPS (Fratini, Pecora, and Cesta 2008), built on top of APSI Timelines Representation Framework (APSI-TRF) (Cesta et al. 2009) to synthesize flexible temporal action plans and revise them according to execution needs, and an executive a la T-REX (Py, Rajan, and McGann 2010) to synchronize the different components under the same timeline representation; (b) functional layer (Bensalem et al. 2010) which combines a state of the art tool for developing functional modules of robotic systems (G$^{en}{}_o$M) with a component based framework for implementing embedded real-time systems (BIP); (c) a simulation environment that accepts the commands generated by the functional layer. GOAC has been successfully tested an iRobot ATRV (called DALA) that provides a number of sensors and effectors (http://homepages.laas.fr/matthieu/robots/dala.shtml) and an ExoMars model on the ESA 3DROV simulator (Poulakis et al. 2008).

This section describes a robotic scenario related to the GOAC project exploited as case study for the experimental assessment presented in the experimental section. First, we describe the DALA platform, i.e., the real robotic platform deployed within the GOAC project. Then, we exploit the same scenario in order to show a possible configuration of a control system implemented by means of an APSI Deliberative Reactor. However, independent of our current use, the work described in this paper is valid for any generic layered control architecture (e.g., (Gat 1997)) that integrates a temporal planning and scheduling system.

The Robotic Platform

The DALA rover is one of the LAAS-CNRS robotic platforms that can be used for autonomous exploration experiments. In particular, it is an iRobot ATRV robot that provides a number of sensors and effectors. It can use vision based navigation (such as the one used by the Mars Exploration Rovers Spirit and Opportunity), as well as indoor navigation based on a Sick laser range finder. Then, the use of DALA in the GOAC project was to simulate a robotic scenario as close as possible to a planetary exploration rover.

In this regard, DALA can be considered as a fair representative for a planetary rover equipped with a Pan-Tilt Unit (PTU), two stereo cameras (mounted on top of the PTU), a panoramic camera and a communication facility. The rover is able to autonomously navigate the environment, move the PTU, take high-resolution pictures and communicate images to a Remote Orbiter. During the mission, the Orbiter may be not visible for some periods. Thus, the robotic platform can communicate only when the Orbiter is visible. The mission goal is a list of required pictures to be taken in different locations with an associated PTU configuration. A possible mission actions sequence is the following: navigate to one of the requested locations, move the PTU pointing at the requested direction, take a picture, then, communicate the image to the orbiter during the next available visibility window, put back the PTU in the safe position and, finally, move to the following requested location. Once all the locations have been visited and all the pictures have been communicated, the mission is considered successfully completed. The rover must operate following some operative rules to maintain safe and effective configurations. Namely, the following conditions must hold during the overall mission: While the robot is moving the PTU must be in the safe position (pan and tilt at 0); The robotic platform can take a picture only if the robot is still in one of the requested locations while the PTU is pointing at the related direction; Once a picture has been taken, the rover has to communicate the picture to the base station; While communicating, the rover has to be still; While communicating, the orbiter has to be visible. The reader may refer to (Ceballos et al. 2011) for further details.

Configuring the control system for experiments

The GOAC system follows the approach, first proposed in T-REX (Py, Rajan, and McGann 2010), according to which

the control system is realized as the composition of a set of different deliberative reactors. Goal of this paper is to assess a single deliberative reactor. Taking advantage of the flexibility provided by the GOAC framework, a control system is here defined considering only two different reactors i.e., a *Mission Manager* responsible to perform all the deliberative tasks and a *Command Dispatcher* in charge of executing commands and collecting execution feedback.

More in detail, the Mission Manager reactor is designed to provide plans for user requested goals, i.e., requests for (i) scientific pictures in desired locations, (ii) reaching a certain position and (iii) monitoring a certain area. The deliberative reactor can operate with two different planning policies: a *single goal* policy, in which the deliberative reactor plans the goals one after one, following a sort of batch schema; a *all goals* policy, in which the deliberative reactor generates in a unique planning step a solution plan for all the goals. Then, the timelines planned by the Mission Managers are dispatched for execution to the Command Dispatcher reactor that, in turn, encodes the planned values into actual commands for the rover and uses the replies provided by the functional layer to produce observations on the low-level timelines. Thus, each reactor has a specific functional role over different temporal scopes during the mission: the Mission Manager's temporal scope is the entire mission and potentially can take minutes to deliberate; the Command Dispatcher interfaces to the DALA functional layer and requires minimal latency with no deliberation. It is also worth underscoring that the Mission Manager is the only deliberative reactor in this use of the GOAC architecture. The Command Dispatcher is a fully reactive system that interacts with the actual controlled system with no deliberation task involved. In this paper to execute tests, the DALA rover has been simulated by means of a software environment[1] used for testing the control system during the GOAC project and offering the same robotic functional interface as well as fully replicating the physical rover behaviors (i.e., random temporal duration for uncontrollable tasks). The experiments have been ran on a PC endowed with an Intel Core i7 CPU (2.93GHz) and 4GB RAM.

It is worth underscoring OGATE has been designed to directly connect the deliberative layer of a control architecture. So, performing experiments with either simulated or actual robotic platforms is not relevant: OGATE is monitoring the control architecture not the simulated/actual robotic platform. More in general, OGATE allows to connect any layer of the control architecture. For instance, it is possible to directly measure the battery level of the robot from the functional layer (which usually constitutes a critical resource) and consider it while processing a related metric to be included in the evaluation.

Experimental results

This section illustrates the assessment of the performance of GOAC taking advantage of OGATE for performing intensive automated tests for generating and executing plans in the planetary exploration problem introduced above and considering the control system configuration presented in the previous section. Namely, OGATE is exploited to assess the capability of the GOAC system to successfully control the platform in the given domain but also to measure how different degrees of uncertainty affect the control behaviors. To objectively analyze this, the methodology presented above is exploited to characterize and compare the performance of the GOAC system when modifying the behaviors of the deliberative component. In this regard, the assessment objectives and the scenarios over which test the GOAC system are to be defined as a first step.

Here, the evaluation objective is to analyze the performance of the GOAC system focusing on how the two different planning policies of the deliberative component affect the controller performance. In particular, in order to assess the implementation of a *sense-plan-act* cycle in the GOAC controller, the evaluation is to focus on the time spent by the deliberative reactor while elaborating sensor data, generating plans and dispatching commands to the Command Dispatcher. In this regard, the performance analysis should be performed considering different problems and execution scenarios in order to assess how they affect the system from a high level perspective.

Then, different planning/execution scenarios are considered by varying the complexity of the robotic planning problem dimensions and the execution conditions. In particular, the following elements are considered: (1) *Plan Length.* Problem instances are considered with an increasing number of requested pictures (from 1 to 3). (2) *Plan Flexibility.* For each uncontrollable activity (i.e., robot and PTU movements as well as camera and communication tasks), a minimal duration is set, but temporal flexibility on activity termination is considered, i.e., the end of each activity presents a tolerance ranging from 0 to 10 seconds. This interval represents the degree of temporal flexibility/uncertainty introduced in the system. (3) *Plan Choices.* A number of visibility windows spanning from 1 to 4 is considered as increasing opportunities to communicate picture contents. Increasing the number of communication opportunities raises the complexity of the planning problem with a combinatorial effect. In general, among all the generated problems instances, the ones with higher number of required pictures, higher temporal flexibility, and higher number of visibility windows result as the hardest.

To complete the evaluation design step, metrics are to be defined according to the evaluation objectives, i.e., assess the sense-plan-act cycle. In this regard, the metrics to be analyzed are the following: *State processing time*: the time required by the deliberative reactor to analyze the observations collected from the Command Dispatcher (sense phase). *Deliberation time*: the time spent by the deliberative reactor to generate a solution plan for the considered goals (plan phase). *Operational time*: the time spent by the deliberative reactor to dispatch commands for the other reactor to accomplish its own high-level goals. For the above metrics, the following ranges (in seconds) have been considered: [0, 4] for the *state processing time*; [0, 10] for the *deliberation time* and; [0, 20] for the *operational time*. The ranges for the

[1]DALA software simulator courtesy of Felix Ingrand and Lavindra De Silva from LAAS-CNRS.

metrics have been obtained analyzing the results of different executions of the GOAC architecture in the considered scenarios. In the evaluation, all the metrics have the same weights. Finally, in order to evaluate how the different scenarios described above affect the GOAC performance, the quadrants of the circular charts have been set to present the three metrics considering an increasing number of communications opportunities.

In this work, we focus our attention on the metrics defined above but, in general, OGATE allows also the definition of additional metrics such as number of generated goals, mean processing time per goal, number of observations received and processed, etc. that can be considered within the evaluation process with no additional costs but their definition in the system.

In order to perform the tests execution, a suitable GOAC plugin for OGATE has been implemented and adapted in order to monitor and store all the relevant performance information from the internal components of the controller (again, see (Muñoz et al. 2014) for further details). Then, OGATE has been exploited to (i) generate the considered scenarios, (ii) carry out all the different controller executions and (iii) collect performance data from the GOAC. For each execution setting, 10 runs have been performed and average values for the defined metrics are reported.

After the collection of performance information in all the considered scenarios, OGATE is able to generates a report containing a wide set of charts corresponding to different control configurations, planning problem instances and execution settings. Figure 3 shows an excerpt (related to the 3 pictures scenarios) of the charts provided by the OGATE report.

Assessing the information shown in the reports, a first straightforward evidence that can be elicited observing the charts in Figure 3 is that the controller execution is not completed in all the considered scenarios (see Fig. 3-d-e-f in the case of 3 and 4 communication windows). In some cases, even though the deliberative module is able to generate a valid plan, the GOAC controller fails in properly completing its execution. After some further analysis of that specific scenarios, an issue related to a dynamic controllability problem (Morris and Muscettola 2005) in the execution of the corresponding plans has been identified.

For what concerns the planning policy comparison, data in Fig. 3 shows that the *all goals* policy is unable to solve the scenarios with 3 pictures and 3 communication windows for all flexibilities and the ones with 5 and 10 seconds flexibility and 4 communication windows. Meanwhile, the *single goal* policy can solve all scenarios, experiencing execution failures with 3 pictures, 5 seconds flexibility and 4 communication windows, and also with 10 seconds flexibility and 3 and 4 communication windows. It is possible to observe that, the *all goals* policy usually has the best *operational time* values while for the *state processing time* the better scores corresponds to the *single goal* policy. This is a consequence of how the deliberative reactor dispatches the goals: in the *single goal* policy, it dispatches goals one by one, increasing the time spent in this task (*operational time*), while the *all goals policy* do it just once. In opposition, the *single goal* policy requires only to check the states for only one goal (*state processing time*), managing a shorter list of expected states, while the *all goals* policy must manage a list with the expected states for all pictures. Focusing on the central area, we can observe the average execution time in the circular bar. The time is expressed as seconds/degree, and computed considering both the correct executions and the failing ones (those who reach a timeout of 5 minutes). In the center is presented the *Global Score* that summarizes the metrics into a single value. Considering only this value we can see that increasing the flexibility, the performance of the system decreases for both planning policies. Also, the *single goal* policy shows a significative better *GS* than the *all goals* policy.

Further investigating the values for the three metrics in the 3 pictures scenarios, additional consideration can be inferred. First, focusing on the deliberation time, both policies have a (close to) constant time for each number of pictures. Anyway, the *single goal* policy requires less time to generate the plans, being remarkable the increment of the required time for 3 pictures with the *all goals* policy. Considering the difference between the time spent to generate the plan for 1, 2 and 3 pictures employing the *single goal* policy, we observe that the time is not proportional to the number of pictures (the average values for 1, 2 and 3 pictures are 0.61, 1.32 and 2.06 seconds respectively), which is an indication of the possible presence of an anomaly behavior (as every picture is planned independently, it is expected to take the same time for planning). A further investigation has been then performed considering the *single goal* policy in a scenario with 0 seconds flexibility and with 2 communications windows aiming to acquire 5 pictures. A temporal profile has been generated and it is presented in fig. 4 where the X-axis represent the tick count and the Y-axis is the time measured in seconds, and the filled area is the time spent deliberation for each picture. The temporal profile is also part of the data generated by OGATE, and allows to observe that the *single goal* policy presents an important issue when considering an incremental number of goals. In fact, it fails in obtaining the plan for the last picture in this scenario as it requires an increasing amount of time for processing additional requested pictures. Then, it has been possible to figure out that while processing additional goals, the OMPS planner performs some checks about past constraints, which are not relevant for the current planning but strongly increase the time spent in planning.

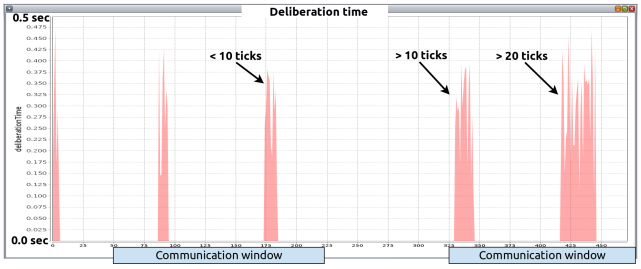
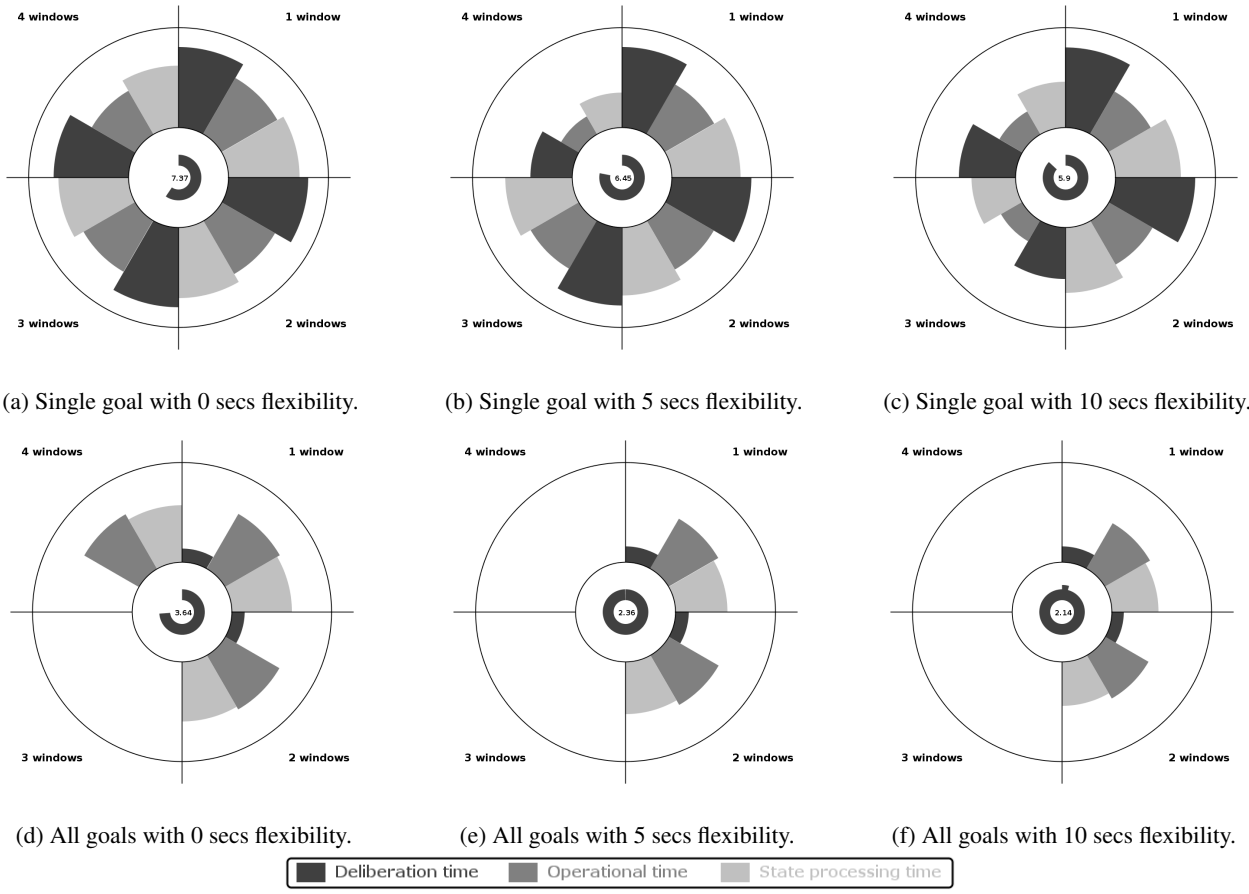


Figure 4: Temporal profiling for the *single goal* policy in a scenario with 5 pictures.

More in general, a detailed analysis of the controller per-



(a) Single goal with 0 secs flexibility. (b) Single goal with 5 secs flexibility. (c) Single goal with 10 secs flexibility.

(d) All goals with 0 secs flexibility. (e) All goals with 5 secs flexibility. (f) All goals with 10 secs flexibility.

Figure 3: GOAC evaluation for 3 pictures with different planning policies and temporal flexibilities of the deliberative component.

formance can be performed by taking advantage of the information provided by the OGATE report. Due to space issue, we report here only a quick overview of the main results. For 1 and 2 communication windows, all the scenarios are correctly completed with both the planning policies. For 3 communications windows, the *all goals* policy is able to complete all scenarios for 2 pictures but it fails to solve the 3 pictures scenarios. The *single goal* policy solves all scenarios for both 2 and 3 pictures (except 3 pictures and 10 secs flexibility) where it fails to execute the scenario 30% times. With 4 communications windows, the *all goals* policy has a worse performance: it solves only 30% and 60% of the scenarios for 2 pictures with 5 and 10 seconds flexibility respectively, and 100% for the scenario with 3 pictures without flexibility. The *single goal* policy solves all scenarios without flexibility; a half in the case of flexibility 5, and, with flexibility 10, it achieves a successful ratio of 90% and 70% for 2 and 3 pictures respectively. As a final consideration, a general observation is that the exploitation of the *single goal* policy in the GOAC architecture seems to be more suitable to address all the considered scenarios but it is affected by the issue shown in Fig. 4. On the other hand, the *all goals* policy allows to overcome such issue but does not provide good performance when facing scenarios with more than two communication windows. So, according to the gathered results, a suitable trade off for deploying the GOAC architecture seems to be the configuration with *single goal* policy when considering a short look-ahead for accepting new picture requests.

Finally, It is worth underscoring how performing the same empirical evaluation without the OGATE support constitutes a significant effort in terms of coding work, customisation of specific metrics to the considered control architecture, collection of performance information and generation of synthetic reports. By using OGATE, the main effort required is related to the implementation of the plugin software for the specific autonomous controller. Thus, the OGATE framework constitutes an off-the-shelf tool capable of performing in automated manner a significant amount of work: that includes the scenarios definition, the tests execution, the collection of information and the report generation.

Discussion and Conclusions

This paper addresses an open issue in autonomous software for robotics platforms: how to perform intensive experiments according to some structured methodology. We have first produced the OGATE framework to support automated testbench campaigns to achieve performance measures of different autonomous controllers. This framework is composed of (i) a methodology to define and guide the testing process, and (ii) an engineering tool to support such a methodology. The paper in particular introduces a method-

ology for evaluating deliberative robotic components and a way for compact visualising the result of its application. Using the new framework we have analyzed different planning policies for the deliberative component of the GOAC system here considered as an external system. Assessment has involved inspecting internal measures of the deliberative component while executing scenarios with increasing complexity. Within these tests we have been able to obtain different reports describing the performance of the system that allows us to obtain conclusions that were hard to be achieved performing standalone tests. In particular for the extensive GOAC testing we have been able to identify an issue in the *single goal* policy of the planning component, and we have been able to conclude that the *all goals* policy is open to dynamic controllability issues.

Some specific issues still require some additional comments. First, the need of considering also stochastic modeling and, more in general, to allow for setting up more comprehensive and complex testing scenarios is a further desired feature. As for assessment of plan-based components, OGATE is a rather new solution and then the present paper is more focused on proposing an evaluation methodology and a technological supporting tool. Indeed, as a starting point we are considering a set of preordered scenarios from a real world robotic application (hence emphasizing the realism of the use case). Nevertheless, OGATE can be seamlessly extended in order to consider more complex scenarios, e.g., including stochastic features and to allow users performing more thorough test campaigns analysis. This is actually an ongoing work.

Then, the choice of relying on plugins enables OGATE to connect to any kind of control architecture notwithstanding which kind of robotic platform/software is deployed. Thus, OGATE aims to be as much as possible platform/software independent. Nevertheless, the implementation of a ROS plugin is in our agenda for future versions of the framework.

In order to develop a plugin to connect OGATE to an autonomous controller, some technical work by skilled engineers and/or control architecture experts is obviously required. Nevertheless, after such mandatory step, OGATE is requiring system users (i.e., not necessarily experts/engineers) to generate two XML files for describing i) the set of components necessary to actually execute the autonomous controller and ii) the metrics to be measured as well as the data required to generate the evaluation reports. Both files can be generated through the graphical environment of OGATE[2]. The file contains detailed information (order of execution, paths, processes, parameters, simulator settings file, etc.) about software processes to be issued in order to properly activate the GOAC and the simulator software.

Finally, among future developments, a further analysis of robot missions evaluation is required to leverage OGATE tool in order to identify relevant metrics to evaluate control architectures also allowing metrics customization according to specific robot mission requirements. In general, the definition of a more thorough set of standard metrics is highly desirable and would constitute a not trivial and important contribution in this field. Moreover, the use of OGATE will be considered for comparing different plan-based deliberative platforms on the same benchmark tests.

[2]As an example, at the following link, an example of an XML file with the information to execute the GOAC controller is provided: https://www.dropbox.com/l/UBKryBFRaZq7taCzdUFCso.

Acknowledgments

Pablo Muñoz is supported by the European Space Agency (ESA) under the Networking and Partnering Initiative. UAH authors are partially supported by the Junta de Comunidades de Castilla-La Mancha project PEII-2014-015-A. CNR authors are partially supported by the Italian Ministry for University and Research (MIUR) and CNR under the GECKO Project (Progetto Bandiera "La Fabbrica del Futuro"). Authors want to thank to the ESA's technical officer Mr. Michel Van Winnendael for his continuous support.

References

Ad Hoc ALFUS Working Group. 2007. Autonomy Levels for Unmanned Systems (ALFUS) Framework – Framework Models. Technical Report 1011-II-1.0, National Institute of Standards and Technology.

Alami, R.; Chatila, R.; Fleury, S.; Ghallab, M.; and Ingrand, F. 1998. An architecture for autonomy. *Field Robotics, Special Issue on Integrated Architectures for Robot Control and Programming* 17:315–337.

Aschwanden, P.; Baskaran, V.; Bernardini, S.; Fry, C.; R-Moreno, M. D.; Muscettola, N.; Plaunt, C.; Rijsman, D.; and Tompkins, P. 2006. Model-unified planning and execution for distributed autonomous system control. In *Association for the Advancement of Artificial Intelligence (AAAI) 2006 Fall Symposia*.

Behnke, S. 2006. Robot competitions – ideal benchmarks for robotics research. In *2006 IEEE/RSJ International Conference on Robots and Systems (IROS) Workshop on Benchmarks in Robotics Research*.

Bensalem, S.; de Silva, L.; Gallien, M.; Ingrand, F.; and Yan, R. 2010. "Rock Solid" Software: A Verifiable and Correct-by-Construction Controller for Rover and Spacecraft Functional Levels. In *i-SAIRAS-10. Proc. of the 10th Int. Symp. on Artificial Intelligence, Robotics and Automation in Space*.

Ceballos, A.; Bensalem, S.; Cesta, A.; Silva, L. D.; Fratini, S.; Ingrand, F.; Ocón, J.; Orlandini, A.; Py, F.; Rajan, K.; Rasconi, R.; and Winnendael, M. V. 2011. A Goal-Oriented Autonomous Controller for Space Exploration. In *ASTRA 2011 - 11th Symposium on Advanced Space Technologies in Robotics and Automation*.

Cesta, A.; Cortellessa, G.; Fratini, S.; and Oddi, A. 2009. Developing an end-to-end planning application from a timeline representation framework. In *IAAI-09. Proc. of the The Twenty-First Innovative Applications of Artificial Intelligence Conference*.

del Pobil, A. P. 2006. Why do we need benchmarks in robotics research? In *2006 IEEE/RSJ International Conference on Robots and Systems (IROS) Workshop on Benchmarks in Robotics Research*.

Fontana, G.; Matteucci, M.; and Sorrenti, D. G. 2014. RAWSEEDS: Building a benchmarking toolkit for autonomous robotics. In Amigoni, F., and Schiaffonati, V., eds., *Methods and Experimental Techniques in Computer Engineering*, SpringerBriefs in Applied Sciences and Technology. Springer International Publishing. 55–68.

Fratini, S.; Pecora, F.; and Cesta, A. 2008. Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. *Archives of Control Sciences* 18(2):231–271.

Gat, E. 1992. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *the Tenth National Conference on Artificial Intelligence (AAAI)*, 809–815.

Gat, E. 1997. On Three-Layer Architectures. In *Artificial Intelligence and Mobile Robots*. MIT Press.

Gertman, D. I.; McFarland, C.; Klein, T. A.; Gertman, A. E.; and Bruemmer, D. J. 2007. A methodology for testing unmanned vehicle behavior and autonomy. In *Performance Metrics for Intelligent Systems (PerMIS'07) Workshop*.

Huang, H.-M.; Messina, E.; Jacoff, A.; Wade, R.; and McNair, M. 2010. Performance measures framework for unmanned systems (PerMFUS): Models for contextual metrics. In *Performance Metrics for Intelligent Systems (PerMIS'10) Workshop*.

Hudson, A. R., and Reeker, L. H. 2007. Standardizing measurements of autonomy in the Artificially Intelligent. In *Performance Metrics for Intelligent Systems (PerMIS'07) Workshop*.

McWilliams, G. T.; Brown, M. A.; Lamm, R. D.; Guerra, C. J.; Avery, P. A.; Kozak, K. C.; and Surampudi, B. 2007. Evaluation of autonomy in recent ground vehicles using the autonomy levels for unmanned systems (ALFUS) framework. In *Performance Metrics for Intelligent Systems (PerMIS'07) Workshop*.

Morris, P., and Muscettola, N. 2005. Temporal dynamic controllability revisited. In *In Procs. of the 20th National Conference on Artificial Intelligence (AAAI-2005)*.

Muñoz, P.; Cesta, A.; Orlandini, A.; and R-Moreno, M. D. 2014. First steps on an on-ground autonomy test environment. In *5th IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*. IEEE.

Nesnas, I.; Simmons, R.; Gaines, D.; Kunz, C.; Diaz-Calderon, A.; Estlin, T.; Madison, R.; Guineau, J.; McHenry, M.; Shu, I.-H.; and Apfelbaum, D. 2006. CLARAty: Challenges and steps toward reusable robotic software. *Advanced Robotic Systems* 3(1):23–30.

Orebäck, A., and Christensen, H. I. 2003. Evaluation of architectures for mobile robotics. *Journal of Autonomous Robots* 14:33–49.

Parasuraman, R.; Sheridan, T. B.; and Wickens, C. D. 2000. A model for types and levels of human interaction with automation. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on* 30(3):286–297.

Poulakis, P.; Joudrier, L.; Wailliez, S.; and Kapellos, K. 2008. 3DROV: A planetary rover system design, simulation and verification tool. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*.

Py, F.; Rajan, K.; and McGann, C. 2010. A Systematic Agent Framework for Situated Autonomous Systems. In *AAMAS-10. Proc. of the 9^{th} Int. Conf. on Autonomous Agents and Multiagent Systems*.

The RoboCup Logistics League as a Benchmark for Planning in Robotics

Tim Niemueller and **Gerhard Lakemeyer**
Knowledge-based Systems Group
RWTH Aachen University, Aachen, Germany
{niemueller, gerhard}@cs.rwth-aachen.de

Alexander Ferrein
MASCOR Institute
FH Aachen, Aachen, Germany
ferrein@fh-aachen.de

Abstract

Planning in robotics as task-level executive is still an exception rather than the norm. Domains are often too dynamic or complex and therefore developers resort to more reactive or deliberative tools. In this paper, we characterize the RoboCup Logistics League (RCLL) as a medium complex robotics planning domain in terms of domain properties, implementation strategies, and planning models. We propose the RCLL in simulation and on real robots as a suitable testbed for a comparison of planning systems.

1 Introduction

Autonomous robots require a task-level executive, a software component that decides on the actions to take to achieve a certain goal. Typical approaches can be roughly divided in three categories: state machine based controllers like SMACH (Bohren and Cousins 2010) or XABSL (Loetzsch, Risler, and Jungel 2006), reasoning systems from Procedural Reasoning Systems (Alami et al. 1998a) or rule-based agents (Niemueller, Lakemeyer, and Ferrein 2013) to more formal approaches like GOLOG (Levesque et al. 1997), and finally planning systems with varying complexity and modeling requirements. There are also hybrid systems integrating aspects of more categories like integrating PDDL-based planning into GOLOG (Claßen et al. 2012).

Planning systems are still the exception rather than the norm in robotics applications, with notable exceptions like (Dornhege and Hertle 2013). Often domains are either too dynamic requiring prohibitively frequent decision points (e.g., robot soccer), or are highly complex imposing tedious modeling requirements to cover (a suitable subset of) the domain (e.g., domestic service robots). In this paper, we propose the *RoboCup Logistics League (RCLL)* as a suitable testbed for a wide variety of planning methodologies. It is a *medium complex domain* inspired by problems from in-factory logistic applications, where a group of robots has to maintain and optimize a material flow among processing machines and eventually deliver to an exit gate. We characterize the domain in terms of classic *domain properties* as a combined cooperative and competitive, dynamic, and continuous environment with partial observability and non-deterministic actions – but the latter two can be relaxed. We describe possible *implementation and modeling strategies* in terms of planning scope and horizon, output type, and distribution, and their implication on suitable *planning models*. The domain contains limited uncertainty in terms of orders posted at times unknown a priori, machines being out-of-order at some times, and due to robots of the other team.

A simulation (Fig. 3) of the competition (Zwilling, Niemueller, and Lakemeyer 2014) is available as open source software.[1] It supports a high-level abstraction that allows to keep the development and integration effort required to implement a task-level executive and a possible later transition to actual robots manageable even for small teams.

2 The RoboCup Logistics League

RoboCup (Kitano et al. 1997) is an international initiative to foster research in the field of robotics and artificial intelligence. It serves as a common testbed for comparing research results in the robotics field. RoboCup is particularly well-known for its various soccer leagues. In 2012, the new industry-oriented RoboCup Logistics League Sponsored by Festo (LLSF) was founded to tackle the problem of production logistics and renamed to RoboCup Logistics League (RCLL) in 2015. Groups of three robots have to plan, execute, and optimize the material flow in a smart factory scenario and deliver products according to dynamic orders. Therefore, the challenge consists of creating and adjusting a production plan and coordinate the group of robots (Niemueller et al. 2013a). In 2015, the league introduced actual production machines (Niemueller et al. 2013b).

The RCLL competition takes place on a field of $12\,\mathrm{m} \times 6\,\mathrm{m}$ partially surrounded by walls (Fig. 3). Two teams of up to three robots each are playing at the same time competing for points, (travel) space and time. The game is controlled by the *referee box (refbox)*, a software component which provides agency to the environment by instructing machines on the field. After the game is started, no manual interference is allowed, robots receive instructions only from the refbox and must act completely autonomous. The robots communicate among each other and with the refbox through WiFi. Communication delays and interruptions are common and must be handled gracefully by the robots.

[1] http://www.fawkesrobotics.org/p/llsf-sim/

Figure 1: Play-offs game at the German Open 2015

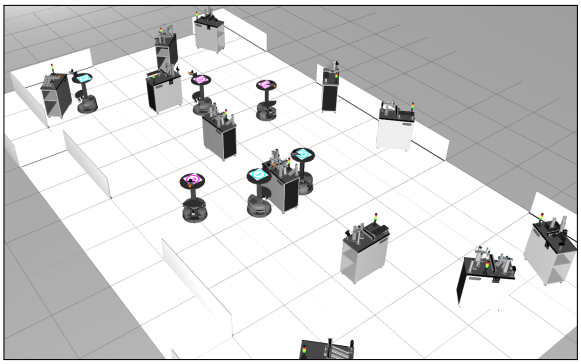


Figure 3: Simulation of the RCLL 2015

Each team has an exclusive set of six machines of four different types of Modular Production System (MPS) stations. The refbox assigns a zone of $2\,\mathrm{m} \times 1.5\,\mathrm{m}$ to each station (position and orientation are random within the zone). Each station accepts input on one side and provides processed workpieces on the opposite side. Machines are equipped with markers that uniquely identify the station and side. They host a signal light to indicate the current status of a machine, such as "ready", "producing" and "out-of-order".

The game is split in two major phases. In the *exploration phase* robots need to roam the environment and discover machines assigned to their teams. For such machines, it must then report the marker ID, position zone, and light signal code. For correct reports, robots are awarded points, for incorrect reports negative points are scored.

In the *production phase*, the goal is to fulfill orders according to a randomized schedule by refining raw material through several processing steps and eventually delivering it. Orders are announced by the refbox. They state the product to be produced, defined as a workpiece consisting of a cup-like cylindrical colored base (red, black, or silver), zero, or up to three colored rings (blue, green, orange, or yellow), and a cap (black or gray). An exemplary production chain for a product of the highest complexity is shown in Fig. 4. All stations require communication through the refbox to prepare and parametrize them for the following production step. Bases are dispensed by the *base station (BS)*. Each team has two *ring stations (RS)* to mount colored rings on a workpiece. Each hosts two unique ring colors. Fig. 2 shows a robot at such a station waiting for a product after mounting a yellow ring. Some ring colors require additional material, that is, one or two additional bases which must be filled into a passive slide (left-most side of the machine in Fig. 2, indicated by diagonally sliced bases in Fig. 4). The refbox determines the affected ring colors randomly once per game. The *cap station (CS)* is used to mount the final cap on a product. It must be pre-filled with a cap first. Only then the buffered cap can be mounted on the actual workpiece. Finally, the finished product must be taken to the *delivery station (DS)*. Points are awarded on successful delivery only. More complex products score higher, but have an increased risk to loose all points if the product is lost due do a handling error late in the production chain.

Figure 2: Robot approaching a ring station in production

The robot used in the competition is the Robotino 3 by Festo Didactic. It features a round base with a holonomic drive, infrared short-distance sensors, and a webcam. Teams are allowed to add additional sensors and computing devices as well as a gripper for workpiece handling. A customized Robotino of the Carologistics Team is shown in Fig. 2. It features an additional laser range finder for self-localization, collision avoidance, and machine alignment, a custom gripper, and two webcams, one for station marker detection and one for light signal recognition. It has an additional laptop to match the computational requirements. It uses the Fawkes Robot Software Framework (Niemueller et al. 2010) for system integration.[2]

To facilitate rapid development we have created a *simulation for the RCLL* based on Gazebo (Zwilling, Niemueller, and Lakemeyer 2014) depicted in Fig. 3. It uses the refbox to provide the same environment agency as a real game and can be used to operate on several levels of abstraction, e.g., providing either laser sensor data for self-localization or the

[2]The full software stack is available as open source on `https://www.fawkesrobotics.org/p/llsf2014-release`

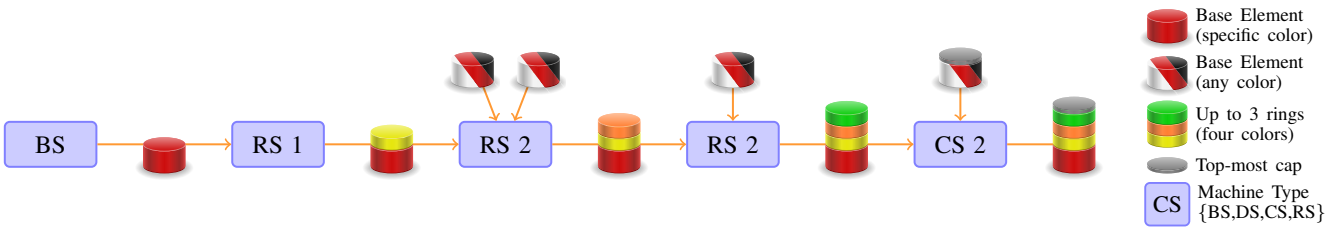


Figure 4: Refinement steps for production of highest complexity product in RCLL 2015 (legend on the right).

position of the robot with noise directly from the simulation. The simulation is crucial for the efficient development of the task-level executive, i.e., a reasoning or planning system that implements and executes the goal-driven behavior of the robots, because it provides faster (semi-automated) testing even if not all lower-level components are available, yet. The simulation software according to the rules for 2014 and 2015 is available as open source software.

3 Characterization of the RCLL

In this section we describe some properties of the RCLL domain and characterize potential planning strategies and assess how the RCLL could fit into common planning models.

3.1 Domain Properties

We describe the RCLL following the standard property matrix of (Russell and Norvig 2010) and some additional common properties of interest.

Cooperative vs. Competitive The RCLL is inherently a *multi-robot and multi-agent domain*. While the game can be played with a single robot, meaningful score can only be achieved by efficient cooperation of the robot group. In a cooperative setting, agents have common interest and try to maximize a common performance measure. In a competitive environment, agents have adversarial goals.

The RCLL scenario is *both*, cooperative and competitive at the same time. Robots of the same team directly cooperate on the task itself. Robots of opposing teams indirectly cooperate by avoiding collisions with each other and not blocking machines of the other team. The setting is still competitive, as both teams try to achieve the highest score. However, the robots compete for travel space only, i.e., while indeed avoiding collisions, there is no need to give way to another robot immediately. Other resources, like production machines, are exclusively assigned to one team and robots must not intentionally disturb each other.

Full vs. Partial Observability Describes if agents can observe all relevant aspects of the world through its sensors or only parts of these. In the latter case, a probability distribution over the possible observations outcomes is desirable.

The RCLL is inherently partially observable. Positions of opponent robots are generally unknown and can only be observed if close to a robot of the own team. Additionally, machine signals can only be read when in front of them, therefore a robot must approach it, for example, to check if a production has completed.

Deterministic vs. Non-Deterministic Determines whether the outcome of actions or observations is completely determined by the current state and the executed action. Actions are called *stochastic*, if a probability distribution over the non-deterministic action outcomes is known. A similar statement can be made about observations, i.e., for non-deterministic one must consider that sensor interpretation might be wrong or imprecise. For stochastic observations a probabilistic sensor model can be specified.

The RCLL is a non-deterministic domain. For example, a product might be lost on the way if the grasp was unstable or dropped when feeding it into a machine. A stochastic model is typically not known for the actions. However, for observations, like recognizing the signal light pattern, models might be determined empirically. Typically, sensor data is aggregated, for example, in a Bayesian form, to reach the desired certainty, balancing time and accuracy.

Episodic vs. Sequential In an episodic environment a robot's experience is divided into atomic episodes each not depending on the actions taken previously. In the sequential case previous action potentially influence future decisions.

The RCLL is clearly sequential, as every decision regarding the task structure, traveled routes, or resource assignment influences the remainder of the game and other robots.

Static vs. Dynamic Classifies whether the world does not change while the agent reasons about it, or if it does in the case of dynamic domains.

The RCLL is dynamic, as robots of the other team make their own independent choices. Even robots of the same team may make decisions while reasoning and the refbox triggers events of the environment, like finishing a production step or setting a machine out-of-order.

Discrete vs. Continuous Determines whether the state of the environment can be described strictly in discrete terms, or if continuous elements exist.

The RCLL can be modeled either way, but leans towards a continuous representation. The game has the concept of a continuous game time and robot motions are continuous. However, the time can be discretized, say, to seconds, as can the positions, for instance, as being at a certain machine, spot on the field, or on the move.

Known vs. Unknown If the environment is known, the outcomes for all actions are given.

The RCLL's environment is known, in particular because its agency is provided for by the referee box. Filling a product into a certain machine will produce a specified outcome depending on whether a valid production step was triggered.

Implicit vs. Explicit Time Distinguishes durative actions with an explicit time span from actions where the duration is only given implicitly.

The times in the RCLL are mostly explicit. For example, a production step at a certain machine requires a certain amount of time. While this time is randomized, it is consistent within a single game and is specified to be in a certain range. It is generally impossible to correctly estimate the duration of actions, as for example, a moving robot might need to slow down to avoid a collision.

Finite vs. Infinite Domain Determines whether the domain of the planning problem is finite or infinite.

That depends on the modeling granularity chosen. It is possible to represent the domain with a finite domain for the general game. But some aspects require infinite domains, for example, if the time is modeled in a continuous fashion or actual robot positions are needed.

3.2 Strategy Characterization

The implementation of a planning and execution system for the RCLL might follow different strategies. They determine the choice of modeling, applicable tools and planners, and system requirements. The criteria are derived from (Alami et al. 1998b).

Global vs. Local Scope Determines whether to consider the overall fleet of robots or to limit reasoning and planning to a single robot.

The decision for the planning scope has a major influence in many regards. First, the granularity of planned actions typically varies. While on a global scope, planning would more likely be about task planning, that is, when is which robot to handle certain machines, but on a local scope individual actions like moving, or grasping a product will probably play a role. Additionally, it determines how much information must be communicated. For a global scope, all information must be available for the planner, either on a robot or on a central station. Then, the tasks must be assigned and communicated to the individual robots. On a local scope, robots need only to communicate the important information, and even if communication breaks down for a limited time (not unlikely in a RoboCup competition where WiFi is used for communication), the robots can still continue operations, even though maybe not optimally. Also, a combined strategy is possible where a global task planner creates an optimal plan given the known information and generates jobs. These could be assigned to the individual robots, for example, by using an auction scheme, which in turn plan individually or employ an approach like plan merging (Alami et al. 1998b).

Complete vs. Incremental Planning Specifies whether to generate a full plan towards a final goal, or whether to plan for some specified horizon and incrementally extend the plan while executing it.

Current typical systems in the RCLL follow an incremental strategy where only the next best option is determined. If a planning system is used, longer planning horizons or even complete plans could be considered.

Centralized vs. Distributed Differentiates whether a centralized instance plans for all robots and communicates plans or if the robots plan for themselves and then coordinate.

In the RCLL, a central station is allowed to provide additional computing power for coordination and planning. But since communication is unreliable, this comes at the risk of not being able to communicate information or task assignments. At this time, all teams follow a distributed scheme. If more capable planning is introduced into the league, a centralized station might provide the required processing power.

Centralized planning could mean that a central station runs individual planners for each robot and then coordinates and distributes these plans. A centralized scheme does therefore not necessarily imply a global planning scope.

Sequential vs. Conditional Plans Determines whether the resulting plan is a sequence of actions or a policy.

For the RCLL, it seems more likely to generate a sequential plan and adapt it or re-plan if the plan is no longer executable, for example, because new information rendered it invalid. On the other hand, the number of decision points in the RCLL should be manageable when generating a policy.

3.3 Planning Models

A planning model (Geffner and Bonet 2013) depends on the domain properties (Sec. 3.1) and its application is influenced by the chosen planning strategy (Sec. 3.2). We will describe the major planning models and our expectation of the applicability for the RCLL. However, the domain has yet to be formally described and therefore some assumptions may not hold in the end.

Classical Planning Classical planning assumes full information about all parameters relevant at planning time and deterministic actions.

Obviously, in a robotics scenario like the RCLL simplifying assumptions are necessary for modeling. Actions do fail occasionally and some information is simply not known, like machines being out-of-order, or processing and order times.

Approaches like Continual Planning – cf. (desJardins et al. 1999) or (Brenner and Nebel 2009) – which interleave planning and execution, can remedy these problems. Then, classical planners can be used to plan for certain run-time assumptions (e.g., all machines are available) and monitor plan execution to recover from failed actions or an unexpected world state by re-planning.

Conformant Conformant planning allows to account for sensor feedback and uncertainty in the environment.

The RCLL should be a suitable domain for conformant planning, as the sensor feedback relevant to planning can be minimized to machine status feedback, e.g., signal light patterns. A conformant plan could thus plan for both possible situations at a machine, it can be working or out-of-order. In the former case the production will be performed, in the latter the planner might decide to move on an perform another production step first.

MDP or FOND If the environment but the actions are non-deterministic, the problem can be modeled as Markov Decision Processes (MDP) in a probabilistic setting, i.e., if

expected action outcomes can be described by a stochastic model, a logical setting is described as Fully Observable Non-Deterministic (FOND).

As we described earlier, the RCLL is not per se fully observable. But the domain description can be relaxed and augmented with assumptions that MDP or FOND planners should be applicable. For example, the planner can simply make assumptions about machine sensor feedback. A controller must then recognize if the machine is indeed out of order and adapt or re-plan.

POMDP or Contingent If an environment is only partially observable, actions are non-deterministic, and there is uncertainty ascribed to sensors, more expressive planning methods are required. Contingent planning extends the classical model with sensing, while partially observable MDPs (POMDP) require a stochastic sensor model.

This class of planners subsumes the previous models, but planners have to cope with a much increased complexity and thus typically require much longer planning times. The RCLL should provide enough complexity to justify the use of these models, but also not overwhelm the domain designer with details that have to be modeled.

4 Challenges for Planning Systems

Planning in the RCLL is applicable in particular in two forms: as a local-scope on-board planning system for the individual robots, and a central global-scope planning system that considers the whole group.

Local-scope planning allows for a wide range of applicable solutions. The basic problem can be simplified to the STRIPS subset (Fikes and Nilsson 1972) of classical planning systems. We have created an example domain for which an excerpt is shown in List. 1. For reasons of brevity, we have omitted actions for moving, for putting and retrieving workpieces to and from machines (similar to many other domains), and for processing at BS, CS, or DS stations (similar to provided RS action), as well as constants and some predicates. The example handles the production and delivery of a product with a single intermediate ring. The problem statement in List. 2 describes the initial situation and the goal to deliver a product consisting of a red base, a yellow ring, and a gray cap. We have run the Fast Downward (Helmert 2006) planning system yielding the plan in List. 3 with a simple blind A* search.

Already this greatly simplified example shows that the complete modeling of the RCLL requires an elaborated domain description. For the full scenario, we are currently investigating a formalization based on the ADL subset (Pednault 1989), which greatly shortens and simplifies the domain description. A general shortcoming of either solution is that they do not account for uncertainty.

Global-scope planning handles creating a plan for the overall group of robots. This model allows for a better optimization in terms of resource (robot) usage. When using durative actions, we envision that this might lead to plans where multiple robots cooperate on quickly handling a single processing step, for example, to deliver on time for orders with a short lead time. This, however, is yet to be explored.

A general challenge is the *dynamicity the environment*. Orders are posted at random times with a specified lead time and an order time window when the product is to be delivered. This requires frequent updating of the group plan. Approaches like Continual Planning (Brenner and Nebel 2009), which interleave planning and execution, could remedy these problems by checking continuously whether a plan is still applicable triggering re-planning if it is not.

Additionally, there are several sources of *uncertainty*. For

```
(define (domain rcll-production-local-strips)
  (:requirements :strips :typing)
  (:types machine workpiece machine-type
          base-color ring-color cap-color)

  (:predicates
    (rs-contains ?m - machine ?r - ring-color)
    (wp-has-ring ?w - workpiece ?r - ring-color)
    (delivered   ?w - workpiece ?b - base-color
                 ?r - ring-color ?c - cap-color)
  )

  (:action process-at-RS
    :parameters (?w - workpiece ?m - machine
                 ?b - base-color ?r - ring-color)
    :precondition
      (and (m-type ?m RS) (proc-at ?w ?m)
           (rs-contains ?m ?r) (wp-has-base ?w ?b))
    :effect (wp-has-ring ?w ?r)
  )
  (:action deliver
    :parameters
      (?w - workpiece ?m - machine
       ?b - base-color ?r - ring-color ?c - cap-color)
    :precondition
      (and (is-at ?m) (holding ?w) (m-type ?m DS)
           (wp-has-base ?w ?b) (wp-has-ring ?w ?r)
           (wp-has-cap ?w ?c))
    :effect (and (delivered ?w ?b ?r ?c)
                 (not (holding ?w)) (holding NONE))
  )
)
```

List. 1: PDDL domain (exc.) constrained to the STRIPS subset for the production and delivery of a workpiece with a single ring

```
(define (problem rcll-production-local-strips-c1)
  (:domain rcll-production-local-strips)
  (:objects w1 - workpiece)
  (:init
    (is-at ANYWHERE) (holding w1)
    (m-type C-DS DS)   (m-type C-BS BS)
    (m-type C-RS1 RS) (m-type C-RS2 RS)
    (m-type C-CS1 CS) (m-type C-CS2 CS)
    (rs-contains C-RS1 RING_GREEN)
    (rs-contains C-RS1 RING_BLUE)
    (rs-contains C-RS2 RING_YELLOW)
    (rs-contains C-RS2 RING_ORANGE)
  )
  (:goal (delivered w1 BASE_RED RING_YELLOW CAP_GREY))
)
```

List. 2: PDDL problem to produce product with red base, a yellow middle ring, and a gray cap (constants in orange)

```
(drive-to anywhere c-bs)
(process-at-bs w1 c-bs base_red)
(drive-to c-bs c-rs2)
(bring-product-to w1 c-rs2)
(process-at-rs w1 c-rs2 base_red ring_yellow)
(get-product-from w1 c-rs2)
(drive-to c-rs2 c-cs1)
(bring-product-to w1 c-cs1)
(process-at-cs-mount
   w1 c-cs1 base_red ring_yellow cap_grey)
(get-product-from w1 c-cs1)
(drive-to c-cs1 c-ds)
(deliver w1 c-ds base_red ring_yellow cap_grey)
```

List. 3: Fast downward solution to the planning problem

example, handling the different stations is simple compared to other mobile manipulation tasks, but yet difficult enough to introduce considerable uncertainty into the domain. For example, producing longer product chains score considerably higher, but bear more uncertainty expressing a higher risk of a workpiece handling error. Additionally, robots of the other team may increase travel costs that can make it difficult to achieve accurate action cost estimates.

5 Conclusion

As of this time, teams participating in the RCLL typically employ a local, incremental, and distributed strategy without planning. That is, robots collect information and classify the current situation and commit to the next action. An example is a CLIPS-based agent system (Niemueller, Lakemeyer, and Ferrein 2013) that collects information in a knowledge base and employs a rule-based reasoning system that models a hierarchical task structure to decide on the next action. A planning system, however, could offer a better ground for optimization of the production for the overall fleet of robots.

As a medium complex domain, the RCLL provides a proper balance between required modeling effort, necessary planning model features, horizon, and complexity, and runtime. It is therefore an interesting planning domain allowing to compare planning systems in the context of multi-robot system in a semi-standardized environment providing limited amounts of uncertainty, partial observability, and nondeterminism in action execution. The domain also fosters embedding of planning systems into robotic executives. Especially, the availability of a simulation system reduces the initial development and integration effort.

Acknowledgments. T. Niemueller was supported by the German National Science Foundation (DFG) research unit *FOR 1513* on Hybrid Reasoning for Intelligent Systems (`http://www.hybrid-reasoning.org`).

We gratefully acknowledge travel funding provided by Festo Didactic to present this work at the workshop on Planning in Robotics at ICAPS 2015 in Jerusalem, Israel.

References

Alami, R.; Chatila, R.; Fleury, S.; Ghallab, M.; and Ingrand, F. 1998a. An architecture for autonomy. *The International Journal of Robotics Research* 17(4).

Alami, R.; Fleury, S.; Herrb, M.; Ingrand, F.; and Robert, F. 1998b. Multi-Robot cooperation in the MARTHA project. *Robotics & Automation Magazine, IEEE* 5(1).

Bohren, J., and Cousins, S. 2010. The SMACH High-Level Executive. *Robotics Automation Magazine, IEEE* 17(4).

Brenner, M., and Nebel, B. 2009. Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems* 19(3).

Claßen, J.; Röger, G.; Lakemeyer, G.; and Nebel, B. 2012. PLATAS – Integrating Planning and the Action Language Golog. *KI - Künstliche Intelligenz* 26(1).

desJardins, M. E.; Durfee, E. H.; Charles L. Ortiz, J.; and Wolverton, M. J. 1999. A Survey of Research in Distributed, Continual Planning. *AI Magazine* 20(4).

Dornhege, C., and Hertle, A. 2013. Integrated Symbolic Planning in the Tidyup-Robot Project. In *AAAI Spring Symposium - Designing Intelligent Robots: Reintegrating AI II*.

Fikes, R. E., and Nilsson, N. J. 1972. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3):189–208.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26.

Kitano, H.; Asada, M.; Kuniyoshi, Y.; Noda, I.; and Osawa, E. 1997. RoboCup: The Robot World Cup Initiative. In *Proc. 1st Int. Conf. on Autonomous Agents*.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1-3).

Loetzsch, M.; Risler, M.; and Jungel, M. 2006. XABSL - A Pragmatic Approach to Behavior Engineering. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Niemueller, T.; Ferrein, A.; Beck, D.; and Lakemeyer, G. 2010. Design Principles of the Component-Based Robot Software Framework Fawkes. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*.

Niemueller, T.; Ewert, D.; Reuter, S.; Ferrein, A.; Jeschke, S.; and Lakemeyer, G. 2013a. RoboCup Logistics League Sponsored by Festo: A Competitive Factory Automation Testbed. In *RoboCup Symposium 2013*.

Niemueller, T.; Lakemeyer, G.; Ferrein, A.; Reuter, S.; Ewert, D.; Jeschke, S.; Pensky, D.; and Karras, U. 2013b. Proposal for Advancements to the LLSF in 2014 and beyond. In *Proc. of 1st Workshop on Developments in RoboCup Leagues at IEEE ICAR*.

Niemueller, T.; Lakemeyer, G.; and Ferrein, A. 2013. Incremental Task-level Reasoning in a Competitive Factory Automation Scenario. In *AAAI Spring Symposium - Designing Intelligent Robots: Reintegrating AI*.

Pednault, E. P. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. of the 1st Int. Conference on Principles of Knowledge Representation and Reasoning*.

Russell, S. J., and Norvig, P. 2010. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education.

Zwilling, F.; Niemueller, T.; and Lakemeyer, G. 2014. Simulation for the RoboCup Logistics League with Real-World Environment Agency and Multi-level Abstraction. In *RoboCup Symposium*.

Active Perception: Using Goal Context to Guide Sensing and Other Actions

Andreas Hofmann, Paul Robertson
Dynamic Object Language Labs, Inc., 114 Waltham St., Lexington, MA, USA
andreas@dollabs.com, paulr@dollabs.com

Abstract

Existing machine perception systems are too inflexible, and therefore cannot adapt well to environment uncertainty. We address this problem through a more dynamic approach in which reasoning about context is used to actively and effectively allocate and focus sensing and action resources. This *Active Perception* approach prioritizes the system's overall goals, so that perception and situation awareness are well integrated with actions to focus all efforts on these goals in an optimal manner. We use a POMDP (Partially Observable Markov Decision Process) framework, but do not attempt to compute a comprehensive control policy, as this is intractible for practical problems. Instead, we employ *Belief State Planning* to compute point solutions from an initial state to a goal state set. This approach automatically synthesizes perception data flow processes, by generating action sequences that optimally combine state-changing actions that achieve a goal state set with supporting sensing actions that reduce uncertainty in the belief state.

Introduction

In most existing machine perception systems, the perception components are statically configured, so that sensor data is processed in the same, bottom-up manner each sensing cycle. The parameters of components in such a system are also statically tuned to operate optimally under very specific conditions. If higher level goals, context, or the environment change, the specific conditions for which the static configuration is intended may no longer hold. As a result, the static systems are prone to error because they cannot adapt to the new conditions; they are too inflexible.

In addition to their inflexibility, existing machine perception systems are often not well integrated into the autonomous and systems to which they provide information. As a result, they are unaware of the autonomous system's overall goals, and therefore, cannot make intelligent observation prioritization decisions in support of these goals. In particular, it may not be necessary or useful for the perception system to be aware of every aspect of a situation, and it may be detrimental, due to resource contention and time limits, to the overall goal.

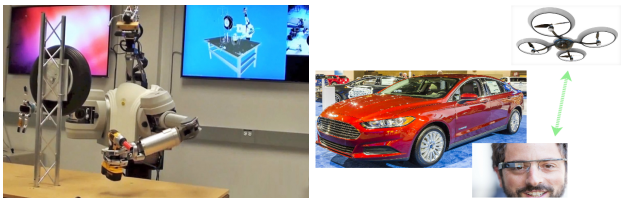
We address these challenges by using a more dynamic *Active Perception* approach in which reasoning about context is used to actively and effectively allocate and focus sensing and action resources. Our system reasons about dynamically changing goals, contexts, and conditions, and therefore, is able to change to a more appropriate process flow configuration, or to better parameter settings in an intelligent way. The *Active Perception* approach (Hofmann and Robertson. 2015; Pineau et al. 2003; Prentice and Roy 2009), prioritizes the system's overall goals, so that perception and state-changing actions are optimally combined to achieve the goals. As a result the approach is more robust to environmental uncertainty.

Active perception draws on models to inform context-dependent tuning of sensors, direct sensors towards phenomena of greatest interest, to follow up initial alerts from cheap, inaccurate sensors with targeted use of expensive, accurate sensors, and to intelligently combine results from sensors with context information to produce increasingly accurate results. The model-based approach deploys sensors to build structured interpretations of situations to meet mission-centered decision making requirements.

Problem Statement

Examples of autonomous systems operating in dynamic and uncertain environments, and that depend on intelligent perception, include cyber defense systems, tactical battlefield advisors, automated transportation and logistics systems, and automated equipment diagnosis and repair systems. Consider, for example, the overall problem of vehicle inspection and repair, and more specifically the problem of changing a tire. This has been a "textbook" problem for PDDL generative planning systems (Fourman 2000), and also a challenge problem in the Defense Advanced Research Project Agency ARM project (Hebert et al. 2012). There are significant challenges in building an autonomous system that can perform this task entirely, or even one that would just assist with the task (Figure 1). Such a system would have to be able to determine whether a tire is actually flat (Figure 2), and what an appropriate sequence of repair steps should be (Figure 3). It would have to be able to solve many subproblems, such as reliably finding a wheel in an image. The system would have to be able to work in many different environments, a great range of lighting conditions, and for a

comprehensive set of vehicle types.



(a) Fully autonomous system using robot. (b) Semi-autonomous advisory system.

Figure 1: Intelligent machine perception would be needed for both a fully autonomous system (a), and a semi-autonomous advisory system (b). The latter might include observation drones, and Google glasses (Bilton 2012) to guide a human user in the repair.

(a) Tire is flat. (b) Tire is under-inflated. (c) Tire is OK.

Figure 2: The perception system must be able to determine whether a tire is really flat, and which one it is.

(a) Wheel chock. (b) Remove wheel.

Figure 3: Repair steps include stabilizing the car (a), getting the spare tire and tools, jacking up the car, removing lug nuts and wheel (b), and installing the new wheel.

We consider, in detail, a subproblem of the tire change problem: reliably finding the wheels of a vehicle in an image. We show how the use of top-down, model-based reasoning can be used to coordinate the efforts of multiple perception algorithms, resulting in more robust, accurate performance than is achievable through use of individual algorithms operating in a bottom-up way. We also show how our model-based framework can be used to address the entire tire change problem.

The general problem we consider can be stated in the following way. Given one or more *agents* operating in an environment, and given that the agents do not directly know the state of the environment, or even, possibly, parts of their own state, and given a goal state for the environment and agents, the problem to be solved is to compute control actions for the agents such that the goal is achieved. In this case, an agent is a resource capable of changing the environment (and its own) state, by taking action. An agent could be a mobile ground robot, a sensing device, or one of many parallel vision processing algorithms running on a cluster, for example. Given that there is uncertainty in the environment state, because it cannot be measured directly, an agent must estimate this state as best as is possible based on (possibly noisy) observations. Similarly, part of the agent's state, such as the functional status of its components, may not be directly measurable, and must be estimated. Based on the agent's best estimate of the current environment state (and its own state), it should take actions that affect the state in a beneficial way (move the state towards the goal). The actions, themselves may have some uncertainty; they do not always achieve the intended effect on the state. The agents must take both state estimate uncertainty, and action uncertainty into account when determining the best course of action. Actions can also have cost. The agents must balance the cost of actions against the reward of reduced uncertainty and progress towards the goal when deciding on actions.

This problem presents significant challenges. First, the overall state space can be very large. Second, the state space is generally hybrid; it includes discrete variables, such as hypotheses for vehicle type, as well as continuous variables, such as position of a wheel. Third, significant parts of the state space may not be directly measurable, and must be estimated based on noisy observations. Fourth, the effect of some actions on state may have uncertainty. Fifth, the agents must take many considerations into account when deciding on actions: they must take into account the uncertainty of the state estimate, the uncertainty of the action effect on the state, the cost of the action, and the benefit of the action in terms of reducing uncertainty and making progress towards the goal.

Note that some actions are performed to improve situational awareness, and some actions are performed to change the state of the agent and/or environment, to achieve an overall goal (for example, jacking up a car so the tire can be changed). The autonomous system should judiciously mix both types of actions so that the situational awareness is sufficient to achieve the goal. In particular, a good sequence of control actions is one that minimizes cost, where cost attributes include both state uncertainty, as well as cost of the action itself. Note, also, that it is typically not necessary for the system to exhaustively resolve all state uncertainty; it just needs to be certain enough to achieve the overall goal.

The problem to be solved can be stated formally as follows. Suppose that the state space is represented by $S = \{S_e, S_a\}$, where S_e is the state space associated with the environment, and, S_a, that associated with the agent. Suppose, further, that the agent can perform a set of actions, A, and make a set of observations, O. A state transition model represents state evolution as a function of current state and action: $T : S \times A \times S \mapsto [0, 1]$. An observation model represents likelihood of an observation as a function of action and current state: $\Omega : O \times A \times S \mapsto [0, 1]$. A reward model represents reward associated with state evolution: $R : S \times A \times S \mapsto \mathbb{R}$. Given an initial state s_0 and goal states s_g, the problem to be solved is to compute an

action sequence $a_0, \ldots a_n$ such that $s_n \in s_g$, and R is maximized. The agent bases its action decisions on its estimates of the state, which in turn, is based on the observations. The state estimates must be sufficiently accurate to support good action decisions. Note that this problem formulation, expressed in terms of a single agent, is easily extended to allow for multiple agents.

Background and Related Work

A *Partially Observable Markov Decision Process* (POMDP) (Monahan 1982) is a useful framework for formulating problems for autonomous systems where there is uncertainty both in the sensing and actions. A POMDP is a tuple $\langle S, A, O, T, R, \Omega, \gamma \rangle$ where S is a (finite, discrete) set of states, A is a (finite) set of actions, O is a (finite) set of observations, $T : S \times A \times S \mapsto [0, 1]$ is the transition model, $R : S \times A \times S \mapsto \mathbb{R}$ is the reward function associated with the transition function, $\Omega : O \times A \times S \mapsto [0, 1]$ is the observation model , and γ is the discount factor on the reward. The *belief state* is a probability distribution over the state variables, and is updated each control time increment using recursive predictor/corrector equations. First, the *a priori* belief state (prediction) for the next time increment, $\bar{b}(s_{k+1})$, is based on the *a posteriori* belief state for current time increment.

$$\begin{aligned} \bar{b}(s_{k+1}) &= \sum_{s_k} Pr(s_{k+1}|s_k, a_k)\, \hat{b}(s_k) \\ &= \sum_{s_k} T(s_k, a_k, s_{k+1})\, \hat{b}(s_k) \end{aligned} \tag{1}$$

Next, the a posteriori belief state (correction) for the next time increment, $\hat{b}(s_{k+1})$, is based on the a priori belief state for next time increment.

$$\begin{aligned} \hat{b}(s_{k+1}) &= \alpha\, Pr(o_{k+1}|s_{k+1})\, \bar{b}(s_{k+1}) \\ &= \alpha\, \Omega(o_{k+1}, s_{k+1})\, \bar{b}(s_{k+1}) \end{aligned} \tag{2}$$

$$\alpha = \frac{1}{Pr(o_{k+1}|o_{1:k}, a_{1:k})} \tag{3}$$

These equations work well given that the models are known, and given that the control policy for selecting an action based on current belief state is known. Unfortunately, computing these, particularly the control policy, is a challenging problem. Value iteration (Zhang and Zhang 2011) is a technique that computes a comprehensive control policy, but it only works for very small problems. An alternative is to abandon computation of a comprehensive control policy, and instead, compute point solutions for a particular initial state and goal state set.

A promising technique for this is *Probabilistic Planning* (Mausam 2012; Hansen and Zilberstein 2001; Bonet and Geffner 2006), which attempts to plan based on the most likely outcomes. We use a variation of this called *Belief State Planning* (Kaelbling and Lozano-Pérez 2013) that is based on generative planner technology (Helmert 2006). A key idea in this technique is the use of two basic actions for a robotic agent: move and look. A move action changes the state of the robot and/or environment; it may move the robot in the environment, for example. A look action is intended to improve the robot's situational awareness.

The move action is specified using a PDDL-like description language.

```
Move(lstart, ltarget)
  effect: BLoc(ltarget, eps)
  precondition: BLoc(lstart, moveRegress(eps))
  cost:  1
```

The variables lstart and ltarget denote the initial and final locations of the robot. The effect clause specifies conditions that will result from performing this operation, if the conditions in the precondition clause are true before it is executed. Cost of the action is specified in the cost clause. The function BLoc(loc, eps) returns the belief that the robot is at location loc with probability at least (1 - eps). The moveRegress function determines the minimum confidence required in the location of the robot on the previous step, in order to guarantee confidence eps in its location at the resulting step:

$$moveRegress(eps) = \frac{eps - p_{fail}}{1 - p_{fail}} \tag{4}$$

where p_{fail} is the probability that the move action will fail.

The look action is specified as follows:

```
Look(ltarget)
  effect: BLoc(ltarget, eps)
  precondition: BLoc(lstart,
                     lookPosRegress(eps))
  cost:  1 - log(posObsProb
                     (lookPosRegress(eps)))
```

The lookPosRegress function takes a value eps and returns a value eps' such that, if the robot is believed with probability at least 1 eps' to be in the target location before a look operation is executed and the operation is successful in detecting ltarget, then it will be believed to be in that location with probability at least 1 eps afterwards.

$$lookPosRegress(eps) = \frac{eps(1 - p_{fn})}{eps(1 - p_{fn}) + p_{fp}(1 - eps)} \tag{5}$$

where p_{fn} and p_{fp} are the false negative and false positive observation probabilities.

In terms of the POMDP belief state update, the move action corresponds to the predictor (Eq. 1), and the look action corresponds to the corrector (Eq. 2).

For the observations, we make extensive use of two types of feature detection algorithms: *Speeded Up Robust Features* (SURF) (Bay et al. 2008), and *Hough Transforms* (Duda and Hart 1972). Neither of these algorithms, used individually, is satisfactory for solving the wheel detection problem robustly. However, when used together, in an Active Perception framework, they beneficially reinforce each others' hypotheses, allowing for more reliable performance.

ApproachAndImplementation

When evaluating approaches to this problem, it is useful to consider what an architecture for the sensors and state estimation components should look like if it were designed by a human expert (Figure 4). Each sensing algorithm can use the current belief state as input, and can also adjust belief state as output.

This organization based on actions, observations, and belief state fits well with the POMDP formalism. As stated previously, we avoid value iteration approaches (Zhang and Zhang 2011), and instead compute point solutions. Our approach automatically synthesizes architectures such as the one in Figure 4, and generates action sequences for sensing operations that reduce uncertainty in the belief state, and ultimately achieve the goal state set.

We use three main sensing actions: SURF Match, SURF Match Other Wheel, and Hough Ellipse Match. Each of these actions has preconditions (requirements for current belief state), and post conditions (effects on belief state). SURF Match uses the SURF algorithm to perform a preliminary detection of a wheel in an image. SURF Match Other Wheel attempts to find the second wheel, also using SURF, given that the first wheel has been detected. This action also performs vehicle pose estimation, and refines the prediction of where the wheels are. Hough Ellipse Match uses either a Hough Circle or Hough Ellipse transform algorithm to refine the wheel location estimates.

The sequence of actions is computed using a *generative planner*. Given the high uncertainty in our problem, the traditional approach of generating a plan and executing it in its entirety is not suitable. Instead, we adopt a *receding horizon control framework* in which a plan is generated based on the current belief state, but only the first action of this plan is executed. After the action, the belief state is updated based on observations, and an entirely new plan is generated. The process then repeats with the first action from this new plan being executed. This approach is computationally intensive, but it ensures that all actions are based on the most current belief state.

To implement this receding horizon control framework, we use an architecture consisting of four main components: Executive, Planner, Sensor Actions, and Belief State Updater, as shown in Figure 5. The Executive manages the receding horizon control process. It maintains the current belief state, and the goal state set. At each control loop iteration, it passes these to the Planner. The Planner, if successful, returns a plan consisting of actions that transition the system from the current state to a goal state, if there are no disturbances. The Executive takes the first action from the plan and executes it by dispatching the appropriate sensor operation. The sensor operation produces an observation, which is used to update the belief state. The process then repeats.

The following sub-sections describe the four components. We begin with the Belief State Updater component, as understanding of belief state representation and update is fundamental to our approach. We then describe the Executive, Planner, and Sensor Action components.

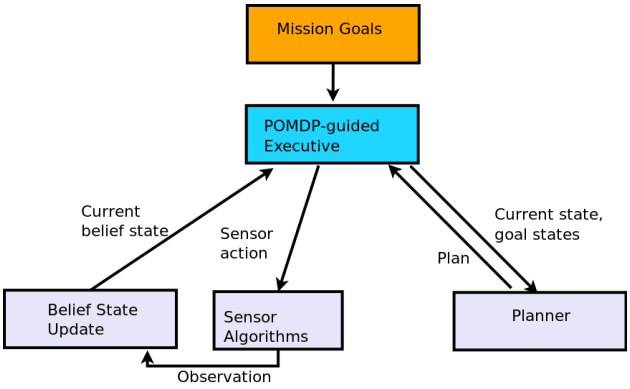


Figure 5: Receding Horizon Control Architecture for Active Perception

Belief State Update

Belief state for a discrete state variable is represented as a *Probability Mass Function* (PMF) over the possible values of the probabilistic variable. Belief state for a continuous state variable is represented using a *Gaussian Probability Distribution Function* (Gaussian PDF) with a specified mean and variance. This could be extended to a representation using a mixture of Gaussians, in order to approximate more complex, non-Gaussian PDFs.

In a traditional POMDP, as explained in the Background section, there are two basic steps for each control iteration. First, an action is taken that modifies the state in some way. The transition function is used to "predict" the effect of the action on the state. Second, an observation is made. The observation function is used to "correct" the prediction of state. The observation function is usually expressed as $\Pr(O|S)$. Thus, it is essentially a static function of state; the observation is performed the same way each time.

Now consider the "move" and "look" methodology described in the Background section. The move action corresponds to the traditional POMDP action, and to the predictor equation (Eq. 1). The look action corresponds to the corrector equation (Eq. 2), but it is also something new; it represents the notion that the observation in the POMDP correction step is, itself, an action. The observation function becomes $\Pr(O|S, A)$, where A is not the action for the move, but rather, a separate "look" action. This action can have parameters: $A\,(p_1, p_2, ...p_n)$, which may be set, intelligently, based on high-level context information. Thus, the observation function becomes something very dynamic, rather than static. This is the key idea behind active perception.

More formally, consider the case where the observation is binary (result is true or false). This is common when the observation is used to test a specific hypothesis, for example, look in the closet to see if the broom is there. We now use Bayes' rule to derive the observation function. In this case, a_1 is the observation action used to confirm the hypothesis that the state is s_1 (if the observation returns true, then the belief in s_1 should increase, if it returns false, belief in s_1 should decrease).

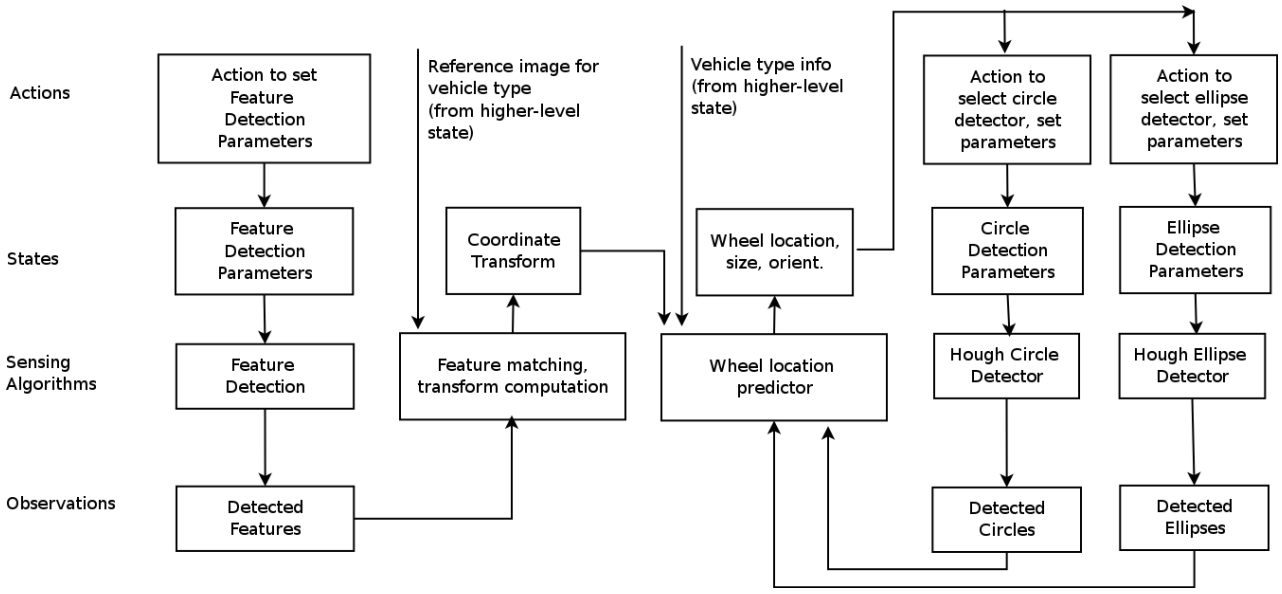


Figure 4: Data flow architecture for wheel finding components.

$$\begin{aligned}&\Pr(s_1|a_1, o = \text{true})\Pr(o = \text{true}|a_1)\\&= \Pr(s_1, o = \text{true}|a_1)\\&= \Pr(o = \text{true}|s_1, a_1)\bar{b}\,(s_1)\end{aligned} \tag{6}$$

Therefore,

$$\hat{b}\,(s_1) = \frac{\Pr(o = \text{true}|s_1, a_1)\bar{b}\,(s_1)}{\Pr(o = \text{true}|a_1)} \tag{7}$$

This is just the standard POMDP observation correction equation (Eq. 2), where $\Pr(o = \text{true}|s_1, a_1)$ is the observation function, and $\Pr(o = \text{true}|a_1)$ is the normalization factor. These terms can be expressed in terms of probabilities of false positives and negatives.

$$\Pr(o = \text{true}|s_1, a_1) = 1 - p_{fn}(s_1, a_1) \tag{8}$$

where p_{fn} is the probability of a false negative given that the state is s_1, and observation action a_1 was taken. Similarly,

$$\begin{aligned}\Pr(o = \text{true}|a_1) = &\Pr(o = \text{true}|s_1, a_1)\bar{b}\,(s_1)\\&+ \Pr(o = \text{true}|s \neq s_1, a_1)(1 - \bar{b}\,(s_1))\\&= (1 - p_{fn}(s_1, a_1))\bar{b}\,(s_1)\\&+ p_{fp}(s \neq s_1, a_1)(1 - \bar{b}\,(s_1))\end{aligned} \tag{9}$$

where p_{fp} is the probability of a false positive. Note that this is the basis for the look regress function (Eq. 5).

Suppose, for example, that the state, s, can have two (discrete) values: s_1 and s_2, and that $\bar{b}\,(s_1) = 0.5$, and $\bar{b}\,(s_s) = 0.5$. Suppose, further, that $p_{fn}(s_1, a_1) = 0.2$, and $p_{fp}(s_2, a_1) = 0.1$. Then, if the observation returns true,

$$\hat{b}\,(s_1) = \frac{0.8 \times 0.5}{0.8 \times 0.5 + 0.1 \times 0.5} = 0.889 \tag{10}$$

and

$$\hat{b}\,(s_2) = 1 - \hat{b}\,(s_1) = 0.111 \tag{11}$$

As expected, belief in s_1 has increased, and belief in s_2 has decreased.

If the observation returns false, then

$$\hat{b}\,(s_2) = \frac{\Pr(o = \text{false}|s_2, a_1)\bar{b}\,(s_2)}{\Pr(o = \text{false}|a_1)} \tag{12}$$

or

$$\hat{b}\,(s_2) = \frac{(1 - p_{fp}(s_2, a_1))\bar{b}\,(s_2)}{(1 - \Pr(o = \text{true}|a_1))} \tag{13}$$

or

$$\hat{b}\,(s_2) = \frac{(1 - 0.1) \times 0.5}{(1 - 0.8 \times 0.5 + 0.1 \times 0.5)} = 0.692 \tag{14}$$

and

$$\hat{b}\,(s_1) = 1 - \hat{b}\,(s_2) = 0.308 \tag{15}$$

As expected, belief in s_2 has increased, and belief in s_1 has decreased.

POMDP Executive

The Executive component implements the top-level receding horizon control loop, coordinating the activities of the planner and sensor components. Algorithm 1 shows pseudocode for the Executive. The algorithm begins by initializing the belief state according to the a priori probabilities, and performing other initialization (Line 1). The receding horizon

control loop begins at Line 5. The first step is to invoke the generative planner in order to determine the next action. A generative planner requires, as one of its inputs, the current state in a deterministic form. The belief state, however, is represented in a probabilistic form, so it first has to be converted to a deterministic form using MostLikelyState (Line 6). The input to the generative planner includes the determinized state, as well as the goal state set. The Executive generates *domain* and *problem* PDDL files, and then executes the planner (Line 7). The planner generates a result file containing a plan, which the Executive reads.

The planner may fail to generate a plan, in which case, the algorithm returns failure. If the planner is successful in generating a plan, the executive dispatches the first action (Line 11). The action (sensor operation) generates an observation. The belief state is updated based on this observation (Line 12). The algorithm terminates if the goal has been achieved, or the maximum number of iterations has been exceeded.

Algorithm 1: `Executive`

Input: a-priori-belief-state-probs, goal-state-set
Output: goal-achieved?

```
        /* Perform initialization.  */
1  belief-state ←
   InitializeBeliefState(a-priori-belief-state-probs);
2  goal-achieved? ← false;
3  max-iterations ← 1000;
4  iteration ← 0;
              /* Begin control loop.  */
5  while not goal-achieved? do
6      current-state
       ← MostLikelyState(belief-state);
7      plan? ←
       GeneratePlan(current-state, goal-state-set);
8      if not plan? then
9          return ;
10     action ← First(plan?);
11     observation ← Dispatch(action);
12     belief-state
       ← UpdateBeliefState(observation);
13     goal-achieved? ←
       CheckGoalAchieved(belief-state, goal-state-set);
14     iteration ← iteration + 1;
15     if iteration > max-iterations then
16         return ;
```

Planner

The Planner component was originally implemented using *Fast Downward* (Helmert 2006), a state of the art generative planner that accepts problems formulated in the PDDL language (McDermott et al. 1998). A limitation of this planner is that it does not support real-valued variables, which necessitated discretization of belief levels (Hofmann and Robertson. 2015). This became unwieldy, so we are now using *Metric FF*, which supports real-valued variables, and linear relations among them. This is an improvement, though the restriction to linear relations requires linearization of the belief update formulas. We have also tried some planners that support non-linear relations, but the ones we tried have not reached a level of maturity to be useable. This is still an important area of ongoing research in the community.

In order for the Planner component to operate properly, it is necessary for it compute belief state updates. This allows it to predict the effect of actions on belief state; a prediction needed for planning purposes. Note that this computation is distinct from the belief state update performed by the Belief State Update component after an observation. The limitations of current planners make it impossible, during planning, to implement completely accurate belief state update, as specified by Eq. 6. Instead, we use an approximation (a linearization in this case), and accept that this approximation is not as accurate as the one performed by the Belief State Update component after an observation. The key point here is that the goal of the Planner component is to make a good choice for the next action, not to predict completely accurately what will happen in the future. The approximation that we use is good enough for our current purposes. More testing is needed to determine whether a more accurate approximation would be beneficial for more complex problems.

We linearize Eq. 7 using a first-order approximation of the form

$$\hat{b}\left(s_1\right) \approx f_0(\epsilon) + f_1(\epsilon)(\epsilon - \bar{b}\left(s_1\right)) \tag{16}$$

where ϵ is the linearization point, the belief value about which Eq. 7 is being linearized, and

$$f_0(\epsilon) = \frac{\left(1 - p_{fn}(s_1, a_1)\right)\epsilon}{(1 - p_{fn}(s_1, a_1))\epsilon + p_{fp}(s \neq s_1, a_1)(1 - \epsilon)} \tag{17}$$

$$f_1(\epsilon) = \frac{df_0(\epsilon)}{d\epsilon} \tag{18}$$

$$\begin{aligned} f_1(\epsilon) = & \frac{p_{fp}(s \neq s_1, a_1)\left(1 - p_{fn}(s_1, a_1)\right)}{\epsilon^2\left(1 - p_{fn}(s_1, a_1)\right)^2} \\ & +2\epsilon\left(1 - p_{fn}(s_1, a_1)\right) p_{fp}(s \neq s_1, a_1)(1 - \epsilon) \\ & +p_{fp}(s \neq s_1, a_1)^2(1 - \epsilon)^2 \end{aligned} \tag{19}$$

We currently use a single linearization, with ϵ manually chosen to be 0.5, the average probability. Using multiple linearizations, with different ϵ values, would allow for approximating the nonlinear belief state update more accurately, with piecewise linearizations.

A PDDL problem formulation consists of a *domain* file, and a *problem* file. The domain file specifies types of actions that can be used across a domain of application such

as logistics, manufacturing assembly, or in this case, finding wheels in an image. The domain file is fixed; it does not change for different problems within the domain. Thus, this part of the formulation was generated manually, and is not modified by the Executive. The problem file, on the other hand, contains problem-specific information such as initial and goal states. Therefore, it must be generated specifically for any new problem. The Executive generates this file automatically, based on knowledge of the goal and belief states. The following PDDL domain file fragment shows the definition of the SURF Match action in PDDL.

```
(:action SURF-match
  :parameters
    (?w ?p ?bsv)
  :precondition
    (and (belief-state-variable ?bsv)
         (pose ?p) (wheel ?w)
         (for-wheel ?w ?bsv)
         (at-pose ?p ?bsv)
         (> (belief-level ?bsv) 0.1))
  :effect
    (and (increase
           (belief-level ?bsv)
           (- (+ (f0)
                 (* (f1)
                    (- (eps)
                       (belief-level ?bsv)))
              (belief-level ?bsv)))
         (increase
           (total-cost)
           (feature-observation-cost ?p))))
```

The precondition clause specifies that the belief state variable value for a particular pose and wheel must be at a minimum of 0.1 in order for this operation to be tried. The effect clause specifies that the belief level increases according to Eq. 16. The cost for the operation is also added to the total cost. The other sensor actions are specified in the PDDL domain file in a similar manner.

Sensor Actions

We now describe in more detail the three sensor actions: SURF Match, SURF Match Other Wheel, and Hough Ellipse Match. The SURF Match action uses the SURF (Speeded Up Robust Features) algorithm (Bay et al. 2008) to attempt to identify wheels by matching a wheel in a reference image with a wheel in the target image. The SURF algorithm is scale invariant, but is somewhat sensitive to large changes in orientation. Therefore, multiple reference images are used, including ones for different orientations (Figure 6). The orientations in the reference images correspond to the orientations of the discrete belief state variables.

The SURF Match action is not highly reliable; it can miss detecting a wheel, especially if the target image is noisy, and it can falsely detect objects that are not wheels. Also, due to the symmetry of wheel images, the SURF Match algorithm does not provide a very accurate estimate of the pose (position and orientation) of the wheel in the target image. Thus, the SURF Match action is used to attempt to achieve a rough initial match, the goal being to move from low to medium confidence estimates, and set the stage for the use of the other sensor actions to improve the estimates.

(a) +2 (b) +1 (c) 0 (d) -1 (e) -2

Figure 6: Wheel reference images, corresponding to different orientations.

The SURF Match Other Wheel sensor action is similar to the SURF Match action, but assumes that one wheel position estimate already exists. It uses this information, along with a number of gists (assumptions), to try to find the other wheel. In particular, it is assumed that the action is observing the left side of the car, and that the car is on level terrain.

If the SURF Match Other Wheel action is successful in finding the other wheel, then it uses the position estimates of both wheels to estimate the pose of the car. The sensor action uses projective geometry, combined with a number of additional gists, to estimate the positions of each wheel in the world coordinate frame, given the image position estimates. These gists are: 1) the height of the camera; 2) the focal length of the camera; and 3) the size of the wheel. All of these are reasonable gists; a ground robot or UAV would know the focal length of its camera, as well as its height. Given previous gists for vehicle type, the size of the wheel can be determined from the vehicle type's spec data.

Once the position estimates of each wheel in the world coordinate frame are known, simple trigonometry is used to determine orientation of the car, particularly, its yaw (rotation about the vertical axis). This estimate is the continuous counterpart to the discrete belief state variables for orientation. The continuous and discrete variables inform and reinforce each other as part of the belief state update mechanism.

The Hough Ellipse Match sensor action uses Hough transforms (Duda and Hart 1972) to determine wheel position in an image with a high degree of accuracy. This action is computationally expensive, but has the potential to give very accurate estimates, when supplied with good parameters. Thus, this action is used after the other, less expensive sensor actions have developed a good hypothesis about the wheel pose.

The computational expense of the Hough transform algorithm rises as the number of parameters increases. Therefore, the ellipse variant is more expensive than the circle variant. For this reason, the Hough Ellipse Match sensor action first checks whether estimated orientation (yaw angle) of the car is small, indicating that the car side is facing the camera directlym ub wgucg case the circle variant is used.

Results

Results using Individual Sensor Algorithms

SURF Algorithm Results The SURF algorithm is susceptible to error, particularly when there is a significant difference in orientation between the wheel in the reference and target images. Figure 7 shows a weak match result due to this problem. Figure 8 shows an incorrect match, also due to this problem.

For this reason, our system uses multiple wheel reference images, at different orientations, as shown in Fig. 6. Which to use is informed by the belief state variables, which hypothesize the most likely orientation of the vehicle.

We performed systematic testing with combinations of vehicle orientations (Fig. 9) and wheel reference images (Fig. 6) in order to determine the observation function used in Eq. 6. Fig. 10 shows PMF's for $\Pr(o|s,a)$. In this case, s corresponds to the hypothesized orientation of the vehicle, represented by a target image in Fig. 9, and a corresponds to the look action, represented by a reference image in Fig. 6. The testing involved selecting a target image and reference image combination, running the SURF algorithm, and noting the number of raw SURF matches.

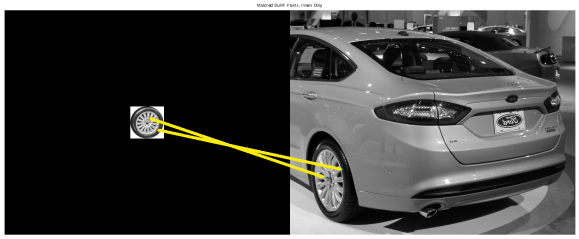


Figure 7: The significant difference in wheel orientation between the reference and target images results in a match of only two points. This does not give a very good estimate of wheel position.

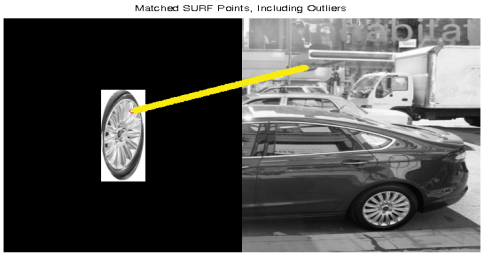


Figure 8: The significant difference in wheel orientation between the reference and target images results in a bad match (false positive).

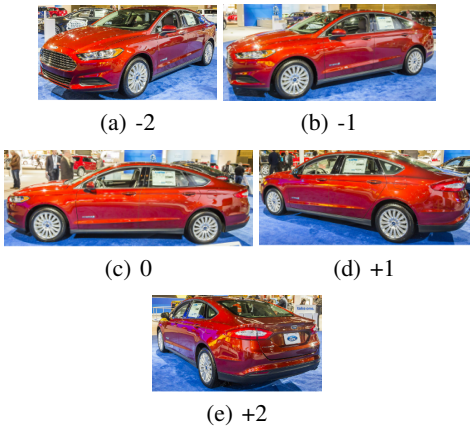


Figure 9: Car target images, corresponding to different orientations.

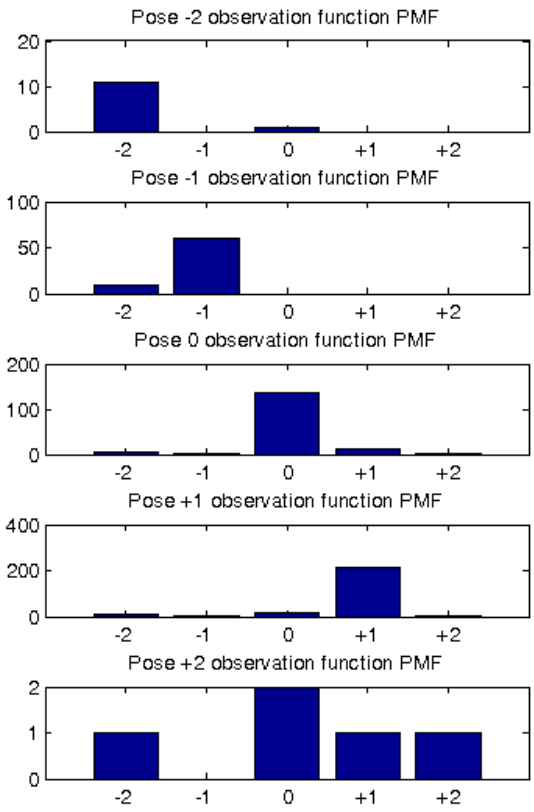


Figure 10: SURF observation function distributions for each look action a, represented by reference image poses -2, -1, 0, +1, and +2. The horizontal axis indicates the target image pose. The vertical axis is number of raw SURF matches. This is easily normalized to get a true PMF. Results for reference image poses -2, -1, 0, and +1 are as expected; the target image with orientation matching the reference image gets the most SURF matches. Results for reference image pose +2 are ambiguous. This is likely due to the extreme angle for this pose.

Hough Ellipse Algorithm Results Figure 11 shows what can easily happen when the Hough Ellipse algorithm is used with insufficient guidance. Instead of finding a wheel, the algorithm has found an elliptical form in the car's grill. Sufficiently constraining the expected ellipse parameters solves this problem.

Figure 11: With insufficient guidance, the Hough algorithm finds an ellipse in the car's grill (highlighted in green), rather than finding the wheel.

The Hough Ellipse algorithm has numerous parameters; how they are set can have a big impact not only on the accuracy of the solution, but also on the time it takes to compute it. First, as shown in Table 1, it is very beneficial to preprocess the gray-scale input image with an edge detection filter, and use the edge detected image as the input to the Hough Ellipse algorithm. Also as shown in Table 1, the algorithm works much faster on scaled images than full-size images. This suggests a staged approach, where small scales are used to obtain an initial solution, and to tightly constrain parameters that are then used in the full size image. Table 1 also shows the beneficial effect on compute time of using the algorithm's randomization parameter. The parameter reduces the number of Hough cells, selecting a smaller number at random. The number of cells used is determined by the parameter. Appropriate settings of this parameter have negligible effect on solution quality, while significantly reducing compute time. Other parameters include rotation span (range of degrees for rotation angle), and bounds on major axis length. Tables 2 and 3 show the benefits of tight bounds for these parameters.

Settings	Image scale			
	full	1/2	1/4	1/8
gray scale	hours	6280	114	4
edge detected	602	14	1.55	0.12
edge detected, randomization	58	5.3	1.7	0.12

Table 1: Hough ellipse algorithm timing results for various image scales, image input types, and randomization settings (times are in seconds). Test image is a wheel shown at an oblique angle, requiring the use of the ellipse rather than the circle version of the algorithm.

Rotation span	Image scale			
	full	1/2	1/4	1/8
40	2515	54.6	6.68	0.36
20	1233	29.5	3.35	0.23
10	537	13.6	1.83	0.16
2	88	2.8	0.33	0.02

Table 2: Hough ellipse algorithm timing results for various image scales, and rotation span settings (rotation spans are in degrees, times are in seconds). Max major axis length = 250, min = 150.

Max	Min	Image scale			
		full	1/2	1/4	1/8
250	150	1233	29.5	3.35	0.23
225	175	500	12.36	1.40	0.14
195	175	299	7.5	0.82	0.085

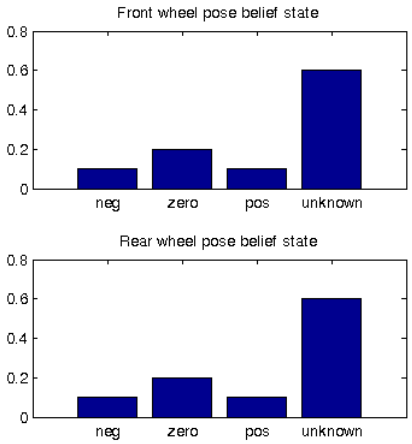
Table 3: Hough ellipse algorithm timing results for various image scales, and max and min major axis length bounds. Axis length bounds are in pixels, times are in seconds. Rotation span = 20 degrees.

Example Test Cases

For the first test case, the target image is as shown in Figure 12. The a priori belief state for the discrete wheel poses is shown in Figure 13 (values for car pose belief state are similar). This indicates that the poses are largely unknown, with a slight bias to the zero pose.

Figure 12: Target image for test case 1.

The first control step iteration, based on this belief state, yields the initial plan shown below (plan 1). The Executive performs the first of these actions, yielding a successful match, as shown in Figure 14. Based on this, wheel pose estimates are updated; the hypothesis for zero pose for the front wheel is strengthened. The second control step iteration, based on this updated belief state, yields plan 2. The Executive performs the first of these actions, yielding a successful match, as shown in Figure 15. Based on this, wheel and car pose estimates are updated to further strengthen the zero pose hypothesis. The third control step iteration, based on this updated belief state, yields plan 3. The Executive performs the first of these actions, yielding a successful match, as shown in Figure 16. In this case, because the pose hypothesis is pose zero (indicating that the camera is directly facing



(a) Wheel

Figure 13: Wheel pose, a priori belief state.

the car), the circle rather than ellipse variant of the Hough transform is used. The history of rear wheel pose belief state values over the control iterations is shown in Figure 17. The zero pose belief increases with successive iterations (observations), whereas the pos and neg pose beliefs decrease.

```
Plan 1
1. SURFMatch(front-wheel pose-zero)
2. SURFMatchOtherWheel(front-wheel pose-zero)
3. HoughEllipseMatch(rear-wheel pose-zero)
Plan 2
1. SURFMatchOtherWheel(front-wheel pose-zero)
2. HoughEllipseMatch(rear-wheel pose-zero)
Plan 3
1. HoughEllipseMatch(rear-wheel pose-zero)
```

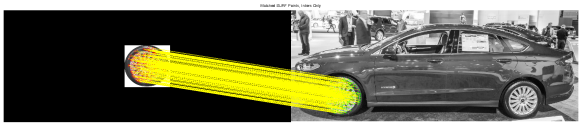


Figure 14: Successful SURF match to front wheel.

Figure 15: Successful SURF match to other (rear) wheel.

For the second test case, the target image is as shown in Figure 18. As before, the planner generates a plan assuming pose zero, based on the a priori belief state. The SURF

Figure 16: Successful Hough ellipse match to rear wheel (match highlighted in green).

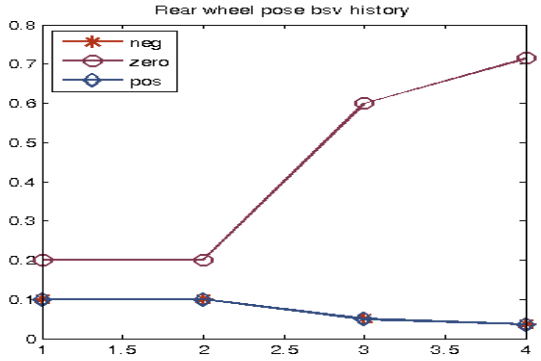


Figure 17: Evolution of belief state for rear wheel pose variable (neg, zero, and pos values).

matches succeed, even though the reference image for pose zero does not exactly match the wheels in the car due to its angle. After the SURF match other wheel action, the pose estimate is improved, resulting in a belief state where pose -1 is most likely. The effect of this new belief state is shown in Plan 3, which uses pose-neg-one, rather than pose-zero as the parameter to the Hough ellipse match action; the circle variation of the algorithm will not work, so it uses the ellipse variation, with bounds on aspect ratio and rotation informed by the car pose estimate. This results in a successful match, as shown in Figure 19. The history of rear wheel pose belief state values over the control iterations is shown in Figure 20. The zero pose belief is initally the highest, but after the SURF match other wheel action (iteration 2), it drops, along with the pos pose belief, while the neg pose belief increases.

```
Plan 1
1. SURFMatch(front-wheel pose-zero)
2. SURFMatchOtherWheel(front-wheel pose-zero)
3. HoughEllipseMatch(rear-wheel pose-zero)
Plan 2
```

```
1. SURFMatchOtherWheel(front-wheel pose-zero)
2. HoughEllipseMatch(rear-wheel pose-zero)
Plan 3
1. HoughEllipseMatch(rear-wheel pose-neg-one)
```

Figure 18: Target image for test case 2.

Figure 19: Successful Hough ellipse match, using ellipse rather than circle variation of the algorithm.

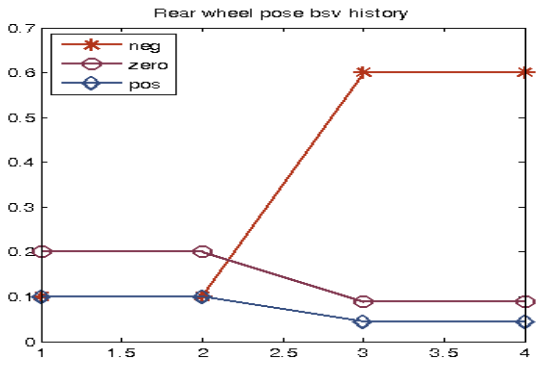


Figure 20: Evolution of belief state for rear wheel pose variable, example test case 2.

Discussion

The focus of our efforts thus far has been on the sub-problem of finding a wheel in an image. This has led to an emphasis on "look" actions, corresponding to the different sensing algorithms (SURF and Hough), and parameterized by which wheel to focus on, and which reference image orientation to use. However, we have not incorporated "move" actions (actions that change the state of the agent or its environment). We believe that the approach we have developed is well suited for incorporating move as well as look actions, with the generative planning component intelligently combining both types of actions. This would allow for testing with more general kinds of problems, where the goal is more than purely a perception goal, but rather, involves achieving an environment goal.

As a next step, we will use a quadcopter platform as a testbed for combining the existing look actions with move actions that move the quadcopter. We expect that this will allow for reliable navigation of the quadcopter around the vehicle, in indoor (garage) environments, while avoiding obstacles like pillars or people. It will also support movement of the quadcopter to close in on a wheel to inspect it more carefully.

Figure 21 shows a grid of navigation waypoints situated relative to a target vehicle. The quadcopter would move between these waypoints in order to navigate, and also to get a better look at the vehicle and wheels. For example, suppose the quadcopter starts at waypoint wp1. A possible plan generated by the planner would mix look and move actions, possibly with the goal of verifying that a wheel actually has a flat tire. The move actions allow quadcopter to move to a more advantageous position, in order to determine with more certainty, the state of the wheel. The look actions determine the state of the wheel, and also keep the target vehicle *anchored* with respect to the quadcopter (Laporte and Arbel 2006; Coradeschi and Saffiotti 2002; Karlsson et al. 2008). This is important for indoor environments, like garages, where GPS navigation is not available.

```
Initial Plan
1. SURFMatch(front-wheel pose-neg-two)
2. Move(wp1, wp2)
3. SURFMatch(front-wheel pose-neg-one)
4. Move(wp2, wp3)
5. SURFMatch(front-wheel pose-zero)
6. HoughEllipseMatch(front-wheel pose-zero)
```

The restriction to linear relations in the generative planner we are currently using allows for a reasonable approximation of belief state update, particularly for the belief state value associated with the look action. However, it does not allow for good normalization across all the values of a belief state variable during planning. Further testing and investigation is needed to determine whether this is a serious shortcoming. In any case, we will investigate planners that allow for nonlinear relations, and therefore, more accurate belief state update during planning. Such planners include sampling-based planners, simple forward heuristic search planners, and SMT solvers.

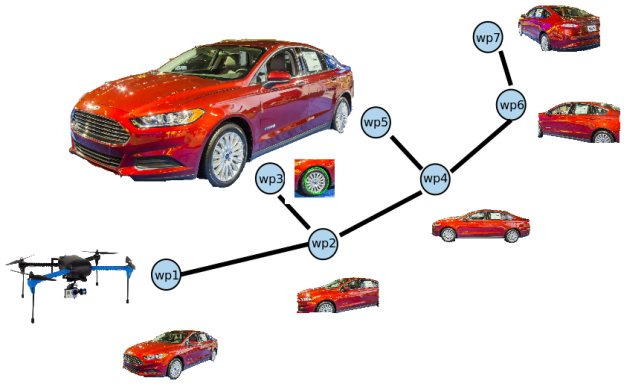


Figure 21: Active perception by quadcopter navigating between waypoints.

Thus far, we have avoided any attempt to learn from all the planning; we are computing point solutions, not learning control policies. While learning comprehensive control policies is generally intractible, it would be interesting to investigate whether partial policies could be learned as a by-product of the planning. A related question is whether a planner, rather than generating a single, rigid plan, could generate a plan with some limited choices. The choices would be made quickly at execution time using a control policy associated with the flexible plan.

Acknowledgments.

This research was developed with funding from the Defense Advanced Research Projects Agency. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

References

Bay, H.; Ess, A.; Tuytelaars, T.; and Van Gool, L. 2008. Speeded-up robust features (surf). *Computer vision and image understanding* 110(3):346–359.

Bilton, N. 2012. Behind the google goggles, virtual reality. *New York Times* 22.

Bonet, B., and Geffner, H. 2006. Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to mdps. In *ICAPS*, volume 6, 142–151.

Coradeschi, S., and Saffiotti, A. 2002. Perceptual anchoring: A key concept for plan execution in embedded systems. In *Advances in Plan-Based Control of Robotic Agents*. Springer. 89–105.

Duda, R. O., and Hart, P. E. 1972. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM* 15(1):11–15.

Fourman, M. P. 2000. Propositional planning. In *Proceedings of AIPS-00 Workshop on Model-Theoretic Approaches to Planning*, 10–17.

Hansen, E. A., and Zilberstein, S. 2001. Lao: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129(1):35–62.

Hebert, P.; Hudson, N.; Ma, J.; Howard, T.; Fuchs, T.; Bajracharya, M.; and Burdick, J. 2012. Combined shape, appearance and silhouette for simultaneous manipulator and object tracking. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, 2405–2412. IEEE.

Helmert, M. 2006. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)* 26:191–246.

Hofmann, A., and Robertson., P. 2015. Active perception: Improving perception robustness by reasoning about context. In *Proceedings of the 10th International Conference on Computer Vision Theory and Applications*.

Kaelbling, L. P., and Lozano-Pérez, T. 2013. Integrated task and motion planning in belief space. *The International Journal of Robotics Research* 0278364913484072.

Karlsson, L.; Bouguerra, A.; Broxvall, M.; Coradeschi, S.; and Saffiotti, A. 2008. To secure an anchor–a recovery planning approach to ambiguity in perceptual anchoring. *Ai Communications* 21(1):1–14.

Laporte, C., and Arbel, T. 2006. Efficient discriminant viewpoint selection for active bayesian recognition. *International Journal of Computer Vision* 68(3):267–287.

Mausam, A. K. 2012. Planning with markov decision processes: an ai perspective. *Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers*.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. Pddl-the planning domain definition language.

Monahan, G. E. 1982. State of the arta survey of partially observable markov decision processes: Theory, models, and algorithms. *Management Science* 28(1):1–16.

Pineau, J.; Montemerlo, M.; Pollack, M.; Roy, N.; and Thrun, S. 2003. Towards robotic assistants in nursing homes: Challenges and results. *Robotics and Autonomous Systems* 42(3):271–281.

Prentice, S., and Roy, N. 2009. The belief roadmap: Efficient planning in belief space by factoring the covariance. *The International Journal of Robotics Research*.

Zhang, N. L., and Zhang, W. 2011. Speeding up the convergence of value iteration in partially observable markov decision processes. *arXiv preprint arXiv:1106.0251*.

Dynamically Extending Planning Models using an Ontology

Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, and **Bram Ridder**
King's College London
London WC2R 2LS
firstname.lastname@kcl.ac.uk

Valerio De Carolis, David Lane, and **Francesco Maurelli**
Herriot-Watt University
Edinburgh, Scotland, UK EH14 4AS
vd63@hw.ac.uk, *d.lane@hw.ac.uk*

Abstract

In this paper we couple a deterministic planner with an ontology, in order to adapt to new discoveries during plan execution and to reason about the affordances that are available to the planner as the set of known objects is updated. This allows us to extend the planning agent's functionality during execution. We use as an example planning for persistent autonomous behaviour in underwater vehicles. Planning in this scenario takes place in a symbolic model of the environment, simulating sequences of possible decisions. Ensuring that the simulation remains robust requires careful matching of the model to the real world, including dynamically updating the model from continuous sensing actions. We describe how our system constructs an initial state for planning, using the ontology; how the ontology is also used to determine the results of each action performed by the planner; and finally demonstrate the performance of the system in a simulation, in which two AUVs are required to cooperate in an unknown environment, demonstrating that with additional reasoning the planning system is able to make new efficient choices, taking advantage of the environment in new ways.

1 Introduction and Motivation

AI Planning [Ghallab *et al.*, 2004] supports a key requirement of intelligent robotics: the ability to perform strategic task-level planning, taking into account limited resources, time, environmental constraints and long-term goals. Planners rely on having access to a rich model of the environment, the objects within it, and the actions that can be performed on the different kinds of objects. A major challenge is that knowledge of the environment constantly changes during plan execution, so the set of objects that can be manipulated cannot be fixed in advance. Instead, the planner's model has to be adapted and updated as new discoveries are made. Acquiring new knowledge in an autonomous way, adapting behaviour accordingly, is a fundamental requirement of persistent autonomous behaviour. Path-planning [Lavalle, 2006] is also a fundamental underlying requirement, but we do not address this topic in this paper.

To enable the acquisition and interpretation of data, and inference over the resulting new knowledge, we provide an *ontology* as one of the components of an autonomous plan-based system. This paper addresses using an ontology in the dynamic construction of AI planning models, using underwater mission-planning for AUVs as an example. In the work presented here, the ontology has two roles: to recognise instances of known concepts from sensed data (eg: to be able to distinguish a valve from a weld), and to identify affordances with the newly recognised objects (eg: to recognise that, being a valve, the object can be grasped, turned, etc). In this way, interpretation of sensed data opens up new reasoning choices for the planner.

Our goal is to show that, equipped with task planning and an ontology, an autonomous system can perform long-term operations without human intervention, adapting to discoveries and increasing its functionality over time. Our approach is to plan operations over a horizon, and replan whenever the ontology updates the planning model. We make two main contributions:

1. The use of a temporal planner, rather than a reactive strategy, to control underwater missions, in order to anticipate and avoid problems rather than simply react to them when they cause actions to fail.
2. The use of an ontology to provide object classifications and affordances to the planner, to improve the planner's interaction with the world.

When planning in robotic domains the actions available to the planner correspond to interactions between the robot and recognised objects in the environment, an idea explored by Geib et al. [2006] as Object-Action complexes. If a plan-based intelligent robot is placed in an environment with a fixed model, the robot might not be able to exploit all of its capabilities as objects are not correctly recognised, or are of unknown types that nevertheless afford known interactions. Using ontological reasoning in conjunction with planning, we extend the capabilities of the planner to more closely match those of the robot in the environment.

We consider the problem of autonomous inspection and maintenance of a seabed oil installation. Regular inspection and maintenance of the facility is to be carried out using autonomous underwater vehicles (AUVs) over extended horizons, so the system must be able to deal with unexpected dis-

covery, and be able to model actions that can be performed in the environment in the symbolic language of the planner. The missions that must be undertaken have many temporal constraints and characteristics. For example, depending on how long it takes to achieve a planned task, the timeframe in which other tasks might be completed can be affected. Thus, depending on the importance of tasks, the planner might decide to make more time available for one task rather than another. In cooperative tasks, the planner has to time the behaviours of the cooperating robots so that they coincide at the right locations.

The combination of planning with other modules that model knowledge about the domain has been explored in other contexts. Several approaches to building semantic maps have been developed. Petrick et al. [2013; 2014] address the issue of joining continuous low-level sensor data with planning in a partially-known symbolic representation with sensing actions in contingent planning. This is similar to the problem that we propose to tackle with an ontology module. Where Petrick et al. cope with unknown knowledge at a Planning level, our focus is on using an ontology to reason about discovered objects, and by so doing extend the possible means of interacting with them. Tenorth et al. [2010] and Galindo et al. [2008] focus on combining semantic knowledge with spatial data to form a *semantic map* of the environment. Tenorth et al. in particular deal with attaching semantic information to a spatial map using an ontology in the ROS[1] framework for indoor household tasks. We define a similar, but general approach to linking an ontology with a planner in ROS.

We validate this approach in an under-water mission, which we describe in Section 2, along with discussion related work. In Section 3 we describe the details of our integration between planning and ontology. We describe the simulations with our system and then conclude in Sections 4 and 5.

2 Planning with Ontologies

Ontologies organise knowledge around concept hierarchies and the relationshipsand attributes between these concepts and instances of them. Considerable work has been carried out on developing representational languages based on description logics and inferences over the sentences they record [Gruber, 1993]. In our framework, an ontology is used as a way for the robot to organise the knowledge about the physical world in which the AUVs operate, not just as geometric concepts, but also as richer structures that offer access to affordances expressed as action templates available in a PDDL [Fox and Long, 2003] domain model (actions applicable to objects of the corresponding types).

In this work we use an ontology to detect new objects and reason about their identification and the relations between them. The planner adapts to unexpected features by revising its abstract, deterministic model of the world and replanning to take account of the new features, while relying on robust control and signal processing systems to handle inaccuracy and noise.

[1]Robot Operating System (ROS); http://www.ros.org; last accessed Apr. 2014

Combining task planning and control have been considered in much prior work. Recent works include: the constraint-based temporal planning system, EUROPA-2, in the T-REX framework [McGann *et al.*, 2008; Py *et al.*, 2010]; combining task and motion planning onboard the PR2 robot [Srivastava *et al.*, 2014]; using homotopy classes to guide path planning [Hernández *et al.*, 2011b; 2011a]; generating a coarse plan to initialise the inspection of an unknown hull [Englot and Hover, 2010]; using a plan-based policy to guide an AUV for autonomously tracking the boundary of the surface of a partially submerged harmful algal bloom [Fox *et al.*, 2012] and autonomous underwater maintanence, explored in the context of temporal planning [Cashmore *et al.*, 2013; 2014];.

2.1 Case Study

We consider a context in which two AUVs have to cooperate to accomplish tasks in a long term mission that requires sustained autonomy: the inspection and maintenance of a seabed facility. Some tasks cannot be achieved by a single AUV and coordination between two or more AUVs is required. Our problem scenarios in this paper are set in this context, focussing on a cluster of inspection tasks. Maintenance tasks take place in a dynamic environment; currents will move the robots, visibility might become obscured, and extraneous events, such as sea animals passing by, might interfere with the execution of an action. In order to find plans that are robust to the uncertainty inherent in the environment, we construct domain models that are conservative with respect to resource requirements (e.g. time and energy). The uncertainty is abstracted by embedding it in the resource estimates used in planning the actions. In this way, we can exploit powerful deterministic temporal planning methods, rather than probabilistic methods which, although they model and reason about uncertainty directly, are much less performant.

During the execution of the plan the ontology is continuously updated using parsed sensor data. As a result of some observations and updates to the model of the world, an executing plan can become invalid (some of the assumptions on which it rests can be violated). This situation is identified by the reasoning within the ontology, coupled with the content on the plan, and will trigger a replan. For example, consider the case where an AUV plans to perform an inspection of a structure, and then move on towards a valve panel. While performing the inspection, the sonar data continuously updates the map. Suppose, during execution of the inspection, the path planned between the structure and the valve panel is found to be blocked. This information is given immediately to the planning system, which will find a new path to the valve panel, and might decide to change the details of the structural inspection to better fit the new strategic plan. This is described in more detail in Section 3. It should be emphasised that the replanning carried out in this case is not path planning (although path planning is a part of the problem), but construction of a *task* plan, linking and coordinating actions between AUVs to achieve the goals of the original plan.

In this paper we focus on a mission involving two AUVs, which we call the *structure inspection task*. The objective is for the two AUVs is to inspect a structure. This requires

exploring a set of inspection points, by navigating close by and directing the sonar and imaging equipment towards them. However, if there are pillars in the structure, the AUVs are able to inspect each complete pillar as a single structure, by observing from a greater distance. This action explores a larger area than any single inspection point, but poses some challenges, as the pillars are deep down in the water, so that a light has to be shone upon the target for the observation to be of sufficient quality.

One solution would be to equip a single AUV with both a camera and a light, but in such a case the light would come from the same direction as the camera, and backscatter would compromise the image quality. Instead, one AUV should approach the pillar and shine the light from an oblique angle, while the other AUV approaches the target to make the observation. Note that it is not possible to sequentialise the behaviour of the two AUVs for the pillar inspection task. The concurrency of the behaviours is crucial to the success of the inspection. This task therefore requires temporal coordination of activities, and temporal reasoning is key for the effective planning of this mission. The combination of temporal coordination of two AUVs, the assignment of tasks between them and the timing and ordering of tasks is the role of a task plan and cannot be resolved by path planning, although path planning is necessary in determing which navigation tasks are achievable and by what route.

The AUVs are placed inside an environment with a number of pillars and other structures. Initially the AUVs have little knowledge of the environment. The AUVs and planning system are capable of performing an inspection in an unknown environment by observing dynamically generated inspection points and replanning as new observations invalidate the current plan.

3 Integration

The symbolic model of the world is provided by the ontology though a ROS interface that the planner uses to construct the initial state of the problem. The symbolic representation of the locations that the AUVs can visit, the objects, and the relationships between them are provided by the ontology. This process is described in four parts. First we describe our model for the structure inspection task, and how it is dynamically updated. We discuss how the model recorded in the ontology is delivered as a problem description for the planner. We then show some solutions to examples of these generated planning problems, and finally describe how these plans are executed.

3.1 Model

The objects collated in the ontology are either known a priori or are derived from analysing sensor data that is collected throughout the execution of a plan. The types of objects include 3d-points within the volume bounding the mission region, which has several subtypes (waypoints, inspection points and strategic waypoints), and structures, which includes the subtypes of pillars and one representing the image slice a sonar captures when intersecting a cylinder, which we call a *Circle* (though it might be elliptical, depending on angle, and will be partially occluded by the solid structure).

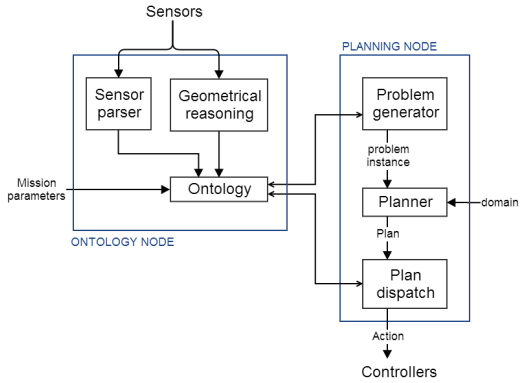


Figure 1: The architecture of integration between the planning system and ontology. The ontology provides the initial state of the planning problem. The ontology is also involved in the exectution of the plan, alerting the planning system when an important part of the environment changes, or is discovered to be different from what is expected.

Figure 1 displays the relationship between the planning system and the ontology.

The possible trajectories the AUVs can traverse are determined by a set of *waypoints*. These waypoints are created using a probabilistic road map (PRM) [Kavraki *et al.*, 1996; Lavalle, 2006], and stored in the ontology. We use an octomap [Wurm *et al.*, 2010] to check if a waypoint collides with an obstacle in the world. Similarly we use the octomap to determine whether the AUVs can traverse between two waypoints. The octomap is built from sensor data and continuously updated.

Inspection points are added to the ontology. These are areas that must be observed by the AUV, either areas of unexplored space, or the unseen sides of possible pillars.

The PRM is augmented with additional waypoints – called *strategic waypoints* – these waypoints are stored in the ontology to provide a denser collection of waypoints around points of interest (in our case the locations of possible pillars and unexplored space). For example, if the sonar picks up a signature that is roughly cylindrical, this is recognised as a new *Circle* and stored in the ontology. New inspection points are inferred as a consequence of the knowledge that such structures can be inspected on all sides. This has the effect of enabling and encouraging the planning system to plan to inspect the opposite side of this object to determine whether it is a *Pillar*.

This information can also be used to 'clean up' the octomap by removing noise. If an object is determined to be a pillar, its shape is known and errors from the sonar can be corrected.

3.2 Constructing the Planning Problem Instance

In our domain, the state of each AUV is partially described by its position, given by a waypoint. The AUV can perform six actions, namely *do_hover_fast*, *do_hover_controlled*, *correct_position*, *illuminate_pillar*, *observe_pillar*, and *observe_inspection_point*, as shown in Figure 2.

```
(:durative-action do_hover_fast
 :parameters (?v - vehicle ?from ?to - waypoint)
 :duration ( = ?duration (* (distance ?from ?to)
         (invtime ?v)))
 :condition (and (at start (at ?v ?from))
         (at start (connected ?from ?to)))
 :effect (and (at start (not (at ?v ?from)))
         (at end (near ?v ?to))))


(:durative-action illuminate_pillar
 :parameters (?v - vehicle ?wp - waypoint ?p - pillar)
 :duration ( >= ?duration 0)
 :condition (and (over all (at ?v ?wp))
        (at start (can_observe_pillar ?v ?wp ?p)))
 :effect (and
        (at start (pillar_illuminated ?p))
        (at start (not (can_observe_pillar ?v ?wp ?p)))
        (at end (not (pillar_illuminated ?p)))
        (at end (can_observe_pillar ?v ?wp ?p))
        (at end (near ?v ?wp))))

(:durative-action observe_pillar
 :parameters (?v - vehicle ?wp - waypoint ?p - pillar)
 :duration ( = ?duration 10)
 :condition (and
        (at start (at ?v ?wp))
        (at start (can_observe_pillar ?v ?wp ?p)))
        (over all (pillar_illuminated ?p))
 :effect (and
        (at start (not (can_observe_pillar ?v ?wp ?p)))
        (at end (observed_pillar ?p))
        (at start (not (at ?v ?wp)))
        (at end (near ?v ?wp))))
```

Figure 2: A fragment of the PDDL inspection-task domain.

The *do_hover_fast* action moves the AUV between two connected waypoints (which, by construction, are the end-nodes of a collision-free edge). Since the fast motion does not take into account final orientation, it only arrives `near` the desired pose. The position must then be corrected. The duration of the action depends on the distance between the two waypoints.

The observe actions allow the AUV to observe an inspection point or a pillar. The precondition requires the AUV to be at a waypoint from which the target inspection point is (partially) visible. Furthermore, the *observe_pillar* action requires the pillar to be illuminated over the whole duration of the observe action. The *illuminate_pillar* action, whose duration is decided by the planner, needs to be performed by a different AUV to meet this requirement.

The problem instance is described using a collection of *objects*, their initial states, and a goal. The objects correspond to object types known by the ontology, and the initial state of the problem instance is generated from the attributes of these objects, also stored in the ontology. This describes the current known state of the world. In the structure inspection task the goal is automatically generated from the initial state, given the current knowledge of the environment. This is done by adding the requirement that every inspection point and pillar has been fully observed. This data is accessed using a ROS interface. In a typical scenario, the goal is initially to observe a set of inspection points $p_1, \ldots, p_n$. When a pillar is discovered, the goal is dynamically updated, and some inspection points $p_j, \ldots, p_k$ are removed from the goal and replaced with the goal of observing the pillar (as it subsumes multiple

```
(define (problem inspection-task-p1)
 (:objects
       auv - vehicle
       wp1 wp2 wp3 ... - waypoint
       ip1 ip2 ip3 ... - inspectionpoint)
       p1 ...          - pillar)
 (:init
  (at auv wp1)
  (= (mission-time) 0)
  (= (observed ip1) 0)
  (connected wp1 wp2) (connected wp2 wp1)
  (= (distance wp1 wp2) 7.16958)
  (= (distance wp2 wp1) 7.16958)
  ...
  (cansee auv ip4 wp12)
  (= (obs ip4 wp12) 0.445331)
  ...
 )
 (:goal (and (>= (observed ip1) 1)
             (observed_pillar p1)
             ...
 ))
 (:metric minimize (total-time)))
```

Figure 3: A fragment of the PDDL inspection-task problem instance.

inspection points, $p_j, \ldots, p_k$, as determined by the appropriate geomtric reasoning in the ontology). Figure 3 shows a fragment of a problem instance.

3.3 Solving the Planning Problem

To solve the problem, we use the temporal planner POPF [Coles *et al.*, 2010]. As described earlier, the planner deals with coarse-grained events: in this case movement between waypoints and observation of inspection points. Example plans in PDDL representation are shown in Figures 4 and 5. In both plans, the AUVs are explicitly given concurrent activities and the planner minimises the duration of the plans. However, in the second case, the plan *requires* concurrency to allow the correct illumination of the pillar during the (long range) inspection task. Note that the illumination duration has been set by the planner to meet the demands of the inspection task.

3.4 Execution

The controllers are responsible for achieving the actions and providing feedback. There are two possible reasons for replanning:

1. *action failure*: an action execution reports failure, using the ROS action feedback, or times out; and
2. *change of environment*: the ontology notifies the planner of a change in the environment that invalidates the plan, or new information, such as new object instances, pertinent to mission goals.

A single plan governs both AUVs. We make the assumption that when replanning, the vehicles can coordinate and share information as required. In practice, this communication is difficult, and in future work we will consider how the plans execution and replanning requests can be coordinated between independent vehicles.

Once the planner has found a plan, the actions are converted into ROS messages and sent to the AUVs. This is done by tokenizing the plan (e.g. figures 4 or 5) and passing the

Without knowledge of Pillars		
Plan time	PDDL action	duration
0.000:	(correct_position auv0 wp_auv0)	[10.000]
0.000:	(correct_position auv1 wp_auv1)	[10.000]
10.001:	(do_hover_fast auv1 wp_auv1 s16)	[46.469]
10.001:	(do_hover_controlled auv0 wp_auv0 s0)	[14.274]
24.276:	(observe_inspection_point auv0 s0 i0)	[10.000]
34.277:	(correct_position auv0 s0)	[10.000]
44.278:	(do_hover_controlled auv0 s0 s1)	[16.971]
56.471:	(correct_position auv1 s16)	[10.000]
61.250:	(observe_inspection_point auv0 s1 i1)	[10.000]
66.472:	(observe_inspection_point auv1 s16 i16)	[10.000]
71.251:	(correct_position auv0 s1)	[10.000]
76.473:	(correct_position auv1 s16)	[10.000]
81.252:	(do_hover_controlled auv0 s1 s2)	[16.971]
86.474:	(do_hover_controlled auv1 s16 s20)	[15.000]
98.224:	(do_hover_fast auv0 s2 s10)	[66.000]
101.475:	(observe_inspection_point auv1 s20 i20)	[10.000]
111.476:	(correct_position auv1 s20)	[10.000]
121.477:	(do_hover_controlled auv1 s20 s17)	[22.649]
144.127:	(observe_inspection_point auv1 s17 i17)	[10.000]

Figure 4: A PDDL plan for an inspection task, found using POPF. Each action has an associated duration, and expected dispatch time, which may differ from the actual execution. Each waypoint and inspection point is associated with its coordinates, as stored in the ontology. In this plan no pillars have been recognised by the ontology.

With knowledge of Pillars		
Plan time	PDDL action	duration
0.000:	(correct_position auv0 wp_auv0)	[10.000]
0.000:	(correct_position auv1 wp_auv1)	[10.000]
10.001:	(do_hover_fast auv1 wp_auv1 s16)	[46.469]
10.001:	(do_hover_controlled auv0 wp_auv0 s0)	[14.274]
0.000:	(correct_position auv0 wp_auv0)	[10.000]
0.000:	(correct_position auv1 wp_auv1)	[10.000]
10.001:	(do_hover_controlled auv0 wp_auv0 s3)	[10.833]
10.001:	(do_hover_fast auv1 wp_auv1 s16)	[46.469]
20.835:	(do_hover_fast auv0 s3 s20)	[84.546]
56.471:	(correct_position auv1 s16)	[10.000]
75.383:	(illuminate_pillar auv1 s16 pillar2)	[50.000]
105.382:	(correct_position auv0 s20)	[10.000]
115.383:	(observe_pillar auv0 s20 pillar2)	[10.000]

Figure 5: A PDDL plan for an inspection task, found using POPF. Each action has an associated duration, and expected dispatch time, which may differ from the actual execution. Each waypoint and inspection point is associated with its coordinates, as stored in the ontology. In this plan, knowledge of pillars exists in the intial state, allowing faster observation of the structure.

actions to the AUV controllers. The actions are dispatched to the two AUVs *concurrently* as scheduled by the plan.

The AUV controllers provide feedback to the executor. If the action is successful, then at the scheduled time, the next action can be dispatched to that controller. If the action is failed, then replanning is triggered. If an action is taking too long to complete, the action is cancelled by the executor and replanning is triggered.

During the execution of an action the executor may cancel the action if the plan is invalidated or the current action

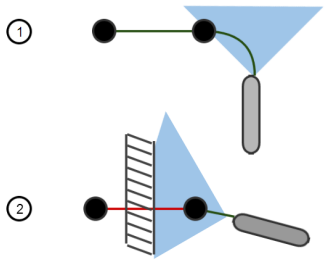


Figure 6: The plan is executed with sonar continuously updating the environment. The plan is invalidated in 2 and replanning is triggered.

is no longer desirable. Note that, in our framework, *replanning* is based on *reformulating* the inspection task as a new planning problem. This re-modelling is performed dynamically, as new information becomes available – the PRM is continuously updated according to the new information about the environment, and the ontology with detected objects and structures to inspect. This reformulation allows the AUVs to adapt to new discoveries during execution.

For example, by collating information continuously it is possible that the obstacle is detected long before the action is to be dispatched. The system can alter the plan before the action is dispatched, thereby avoiding dead-ends and inefficiencies. For example, consider the following simple scenario in figure 6: the AUV is moving between two waypoints, and the sonar detects a wall. The wall obstructs the planned hover action, but will not interfere with the current action. Clearly, replanning should take place to avoid dispatching the doomed action. In this case the obstructed connection is removed from the ontology, the planning system is notified of the change, and finds a new route before dispatching the action.

4 Experimentation

Our objective is to show that the planner and ontology can interact to support the execution of a complex mission. We have designed experiments that demonstrate the following features:

1. Controlled failure of an executing plan in the event of the discovery of new information
2. The discovery of new object instances and their affordances, and updating of the planning problem
3. Replanning of a cooperative mission involving coordinated activity of the two AUVs to achieve mission goals

We show that by continually augmenting the knowledge available, the ontology gives the planner access to previously unknown parts and affordances of the environment leading to plans that are shorter and more efficient than those that can be found without the ontology. We demonstrate the role of planning by considering a mission with interesting temporal structure. The two AUVs must act concurrently in order to inspect pillars. A reactive strategy, in which one or more AUVs patrol the site inspecting pillars as they are encountered, would not be able to arrange coordination of AUV activities and would

# of pillars	IPs only	IPs and Pillars
1	208.909	117.631
1	216.309	135.814
2	365.618	263.028
2	351.014	350.307
3	758.375	383.531
3	781.005	324.062
Avg time (s)	446.872	262.395
Avg # actions	58	23

Table 1: Plan quality for the structure inspection task. "IPs only" refers to the task undertaken without knowledge of pillars. "IPs and Pillars" shows the results when taking knowledge of pillars into account.

result in inefficient inspections. We illustrate this by comparing the behaviour of the AUVs under coordination of the planner with and without the support of the ontology.

The structure inspection missions are carried out in simulation, using a system that emulates an underwater environment and interfaces with ROS. The entire control system has been used to plan and execute missions using physical vehicles, but we have not had the opportunity to test physical missions with two AUVs together. In the test missions, the AUVs are sent to inspect different structures with various numbers of pillars. Initial inspection points are placed on the surface of the structure. Our hypothesis is that using the knowledge from the ontology will allow us to generate plans that have a shorter duration, and fewer actions, because the recognition of a specific instance of an object type gives the planner access to the best affordances to enable efficient interaction with the object. The simulation was run multiple times, first without taking into account knowledge of pillars from the ontology. In this case, all the inspection points had to be observed. Then, the ontological data was taken into account, and pillars could be inspected with the pillar-specific observation action, inspecting multiple inspection points at once.

The times taken to execute the missions are reported. The planning time on each planning cycle is limited to 10 seconds. Figure 7 shows the simulation during runtime. Using the knowledge provided by the ontology about Pillars greatly reduced the time taken to complete the mission (Table 1). Extending the functionality of the planner with new knowledge about the environment can be expected to increase the quality of the plans based on the improvement of affordances.

5 Conclusion

In this paper we describe the linkage of a planning system to an ontology within an execution framework, allowing the planner to exploit features and affordances of elements of the environment as they are identified and inferred by the ontology. As the world state is continually modified using processed sensor data to update the ontology, the executing plan is monitored for validity and replanning is invoked when it ceases to be valid. The result is a system robust to changes in the environment. We tested our system in simulation, showing that using the ontology to associate objects with affordances can result in plans with a shorter duration and fewer actions.

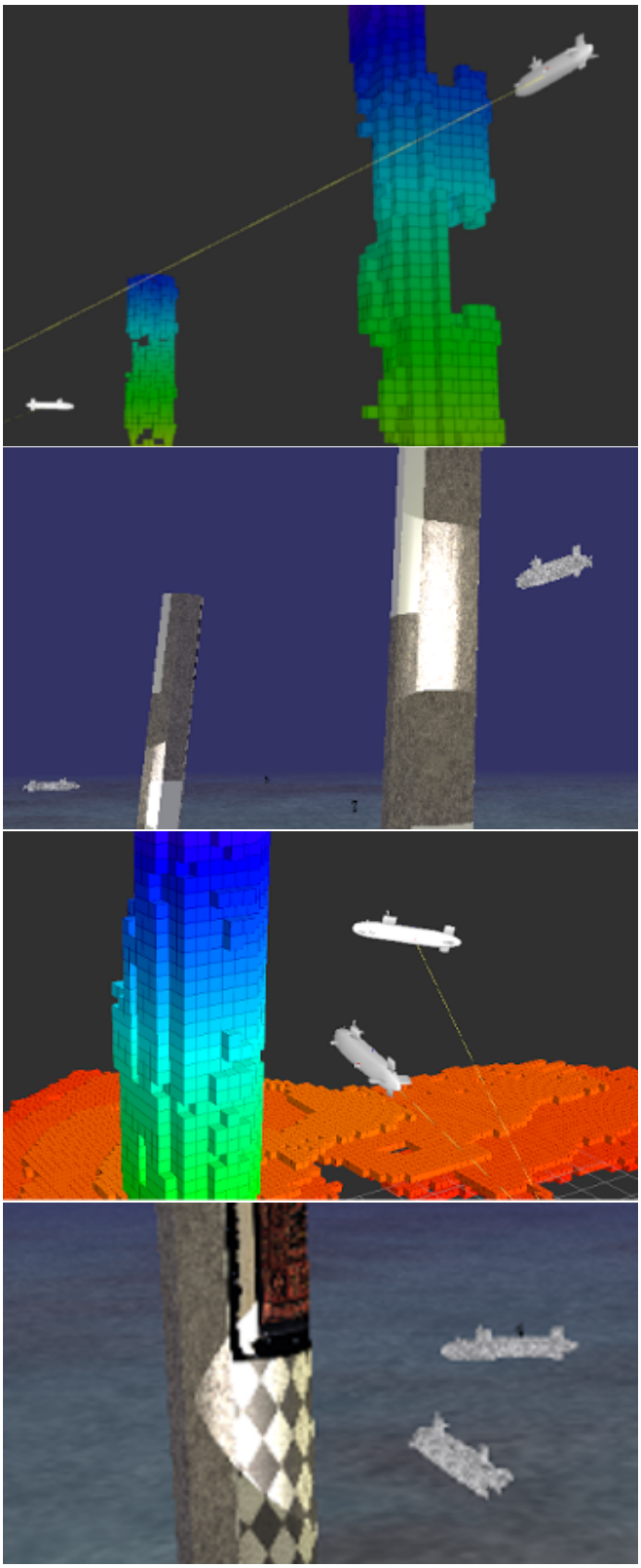


Figure 7: Images of the simulation environment, the 3D model of the AUV structure and sea-bed; and the rviz scene, the environment as detected by the AUV.

References

[Cashmore *et al.*, 2013] Michael Cashmore, Maria Fox, Tom Larkworthy, Derek Long, and Daniele Magazzeni. Planning inspection tasks for auvs. In *Proc. of the MTS/IEEE Oceans 2013 Conference, San Diego (OCEANS'13)*, 2013.

[Cashmore *et al.*, 2014] Michael Cashmore, Maria Fox, Tom Larkworthy, Derek Long, and Daniele Magazzeni. Auv mission control via temporal planning. In *IEEE Int. Conf. on Robotics and Automation (ICRA'14)*, 2014.

[Coles *et al.*, 2010] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In *Proc. of the 20rd Int. Conf. on Automated Planning and Scheduling (ICAPS'10)*, pages 42–49, 2010.

[Englot and Hover, 2010] B. Englot and F. Hover. Inspection planning for sensor coverage of 3D marine structures. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2010.

[Fox and Long, 2003] Maria Fox and Derek Long. PDDL2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Res. (JAIR)*, 20:61–124, 2003.

[Fox *et al.*, 2012] Maria Fox, Derek Long, and Daniele Magazzeni. Plan-based policy-learning for autonomous feature tracking. In *Proc. of the 22nd Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 2012.

[Galindo *et al.*, 2008] Cipriano Galindo, Juan-Antonio Fernandez-Madrigal, Javier González, and Alessandro Saffiotti. Robot task planning using semantic maps. *Robotics and Autonomous Systems*, 56(11):955–966, 2008.

[Geib *et al.*, 2006] Christopher Geib, Kira Mourão, Ron Petrick, Nico Pugeault, Mark Steedman, Norbert Krueger, and Florentin Wörgötter. Object action complexes as an interface for planning and robot control. In *Proc. of the Humanoids-06 Workshop: Towards Cognitive Humanoid Robots*, 2006.

[Ghallab *et al.*, 2004] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[Gruber, 1993] T. R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(1):199–220, 1993.

[Hernández *et al.*, 2011a] Emili Hernández, Marc Carreras, Javier Antich, Pere Ridao, and Alberto Ortiz. A topologically guided path planner for an auv using homotopy classes. In *IEEE Int. Conf. on Robotics and Automation (ICRA'11)*, pages 2337–2343, 2011.

[Hernández *et al.*, 2011b] Emili Hernández, Marc Carreras, and Pere Ridao. A Path Planning Algorithm for an AUV Guided with Homotopy Classes. In *Proc. 21st Int. Conf. on Automated Planning and Scheduling (ICAPS'11)*, 2011.

[Kavraki *et al.*, 1996] L. E. Kavraki, J.-C. Latombe P. Svestka, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE Transactions on Robotics and Automation*, page 566580, 1996.

[Lavalle, 2006] S. M. Lavalle. *Planning Algorithms*. Cambridge University Press, 2006.

[McGann *et al.*, 2008] Conor McGann, Frederic Py, Kanna Rajan, Hans Thomas, Richard Henthorn, and Robert S. McEwen. A deliberative architecture for auv control. In *IEEE Int. Conf. on Robotics and Automation (ICRA'08)*, pages 1049–1054, 2008.

[Petrick and Foster, 2013] Ronald P. A. Petrick and Mary Ellen Foster. Planning for social interaction in a robot bartender domain. In *Proc. of the 23rd Int. Conf. on Automated Planning and Scheduling (ICAPS'13)*, pages 389–397, 2013.

[Petrick and Gaschler, 2014] Ronald P. A. Petrick and Andre Gaschler. Extending knowledge-level contingent planning for robot task planning. In *Proc. of the ICAPS 2014 Workshop on Planning and Robotics (PlanRob)*, pages 157–165, 2014.

[Py *et al.*, 2010] Frederic Py, Kanna Rajan, and Conor McGann. A systematic agent framework for situated autonomous systems. In *Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS'10)*, pages 583–590, 2010.

[Srivastava *et al.*, 2014] Siddharth Srivastava, Eugene Fang, Lorenzo Riano, Rohan Chitnis, Stuart Russell, and Pieter Abbeel. Combined task and motion planning through an extensible planner-independent interface layer. In *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'14)*, 2014.

[Tenorth *et al.*, 2010] Moritz Tenorth, Lars Kunze, Dominik Jain, and Michael Beetz. KNOWROB-MAP - knowledge-linked semantic object maps. In *Humanoids*, pages 430–435, 2010.

[Wurm *et al.*, 2010] Kai M Wurm, Armin Hornung, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: A probabilistic, flexible, and compact 3d map representation for robotic systems. In *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, volume 2, 2010.

Metareasoning for Concurrent Planning and Execution

Dylan O'Ceallaigh and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
dylan.oceallaigh at gmail.com, ruml at cs.unh.edu

Abstract

We consider two fundamental questions in concurrent planning and execution. First, if the domain has an 'identity action' that allows the agent to postpone acting, remain in the same state, and deliberate further, when should this action be taken? Second, if the agent decides to act before a complete plan has been formed, to how many actions in the current partial plan should the agent commit? We show that considering these decisions carefully can reduce the agent's total time taken to arrive at a goal in several benchmark domains, relative to the current state-of-the-art. The resulting algorithm can dynamically adjust the way it interleaves planning and acting, between reactive greedy hill-climbing and deliberative A*, depending on the problem instance.

Introduction

Because of new sensor data, unexpected changes in the world, and execution failures, robots are continually planning and replanning. Yet to interact successfully with humans, robots also need to be responsive and to achieve their goals promptly. They should respond immediately to a goal request and cannot stand still while deliberating endlessly about the perfectly optimal plan. This raises the central challenge of how to organize concurrent planning and execution so that an agent can achieve a goal as quickly as possible while allowing planning and acting to happen concurrently (Ghallab, Nau, and Traverso 2014). In this paper, we formalize a simple version of this problem in the context of real-time heuristic search and address it using an approach based on metareasoning, in which the planner reasons about its own behavior in order to decide when to plan and when to act.

Real-time heuristic search refers to the setting in which a search must finish within a fixed amount of time. Because it may be impossible to find a complete path from the start state to a goal within this time, the search is only required to return the next action for the agent to take. The search is therefore iterated until a goal is reached. Because execution begins before the optimality of the path is verified, no real-time search can guarantee that the agent will follow an optimal trajectory. However, real-time search can nicely handle the realistic setting in which search and execution can happen in parallel: while the agent transitions from state s_i to s_j, the search can plan which action to take from s_j, where the duration of the transition is used as the bound on planning time. This is the setting we address in this paper.

Because we are addressing concurrent planning and execution, we will use goal achievement time (GAT) (Hernández et al. 2012; Burns, Kiesel, and Ruml 2013) as our main evaluation metric. This is the total time taken from the start of the first search iteration to the arrival of the agent at a goal. In a robotics application like fetching a beer for a user, it is natural to minimize this measure, since it corresponds to the time the user has to wait from when the request is issued until it is fulfilled. For an offline algorithm such as A*, this is the sum of planning and execution time. A* will spend a lot of time searching, but then the shortest possible time executing. Because of its real-time constraint, real-time search may execute a longer plan than A*, but because most of its planning will happen concurrently with execution, it may have a shorter GAT than A*. If the cost of an action equals the time taken to execute it, then a real-time search for minimizing GAT can be thought of as minimizing total cost, as usual, as long as the cost of identity actions are taken into account.

Because each search iteration is taking place under a time constraint, a real-time search must be careful about its use of computation time. After performing some amount of lookahead, a real-time search will have identified a most promising state on the search frontier, along with the path leading to it from the current state. It then faces two fundamental questions: 1) should it continue to search, gaining increased confidence that it has identified the correct action to take, or is it ready to begin executing the action that currently appears to be best? In many domains, the agent is allowed to idle in the current state, effectively executing a 'identity' or 'no-op' action while performing additional search. While this deliberation will delay the agent's arrival at a goal, the delay may be worthwhile if the additional planning allows the agent to avoid selecting a poor action. This is perhaps the most fundamental question an agent faces when planning and acting are allowed to interleave or run concurrently. And 2) if the algorithm decides to act, should it commit to all the actions leading to the most promising frontier node, or just some prefix?

Most existing real-time search algorithms decide these questions at design time. For example, the seminal RTA*

algorithm of Korf (1990) always commits to a single action, while state-of-the-art algorithms like LSS-LRTA* (Koenig and Sun 2009) and Dynamic $\hat{f}$ (Burns, Kiesel, and Ruml 2013) always commit to the entire path to the frontier. None of these algorithms take advantage of identity actions. In this paper, we explore whether it can be beneficial to make these decisions dynamically during planning. Considering identity actions, for example, allows a real-time search to decide at runtime whether to behave more like greedy hill-climbing, and commit to an action after only limited lookahead, or behave more like A* (Hart, Nilsson, and Raphael 1968), and explore all the way to a goal before starting to act.

We view our approach as a form of metareasoning, in which the search algorithm governs its own behavior by attempting to estimate the effects of committing to actions versus doing more search. Because the effects of search are necessarily uncertain, the challenge is to efficiently estimate the probability of various outcomes. We believe that the metareasoning perspective provides an elegant and principled way to address the fundamental trade-off between search and execution in planning and combinatorial search.

Concretely, the paper proposes two extensions of the Dynamic $\hat{f}$ algorithm, one addressing identity actions and one addressing prefix commitment. We carefully examine their behavior on a new set of handcrafted pathfinding problems and then run large-scale tests on four benchmark domains. We find that the new techniques successfully allow a real-time search to adapt its behavior between greed and deliberation, matching or outperforming previous state-of-the-art algorithms. This illustrates one principled way in which an agent can appropriately organize concurrent planning and action.

Previous Work

LSS-LRTA*

Local Search Space-Learning Real-Time A* (LSS-LRTA*) is a state-of-the-art real-time search algorithm that has been recommended for situations in which there is an explicit real-time constraint per action (Koenig and Sun 2009). It consists of a two step process. First, LSS-LRTA* performs an A*-like search from the agent's current state toward a goal until some expansion limit is reached. Second, LSS-LRTA* selects the best state on the search frontier, queues the best path to that state for the agent to execute, much as A* does for the complete path to the goal, and proceeds with a Dijkstra-like backup process where the h value of each state in the lookahead search space is updated to the h value of its best child plus the edge cost between the two. This intensive learning step allows the algorithm to escape local minima much faster than previous real-time search algorithms.

Dynamic $\hat{f}$

Dynamic $\hat{f}$ makes two small modifications to LSS-LRTA*. A pseudocode sketch is shown in Figure 1. First, instead of using a fixed time bound, Dynamic $\hat{f}$ sets the time bound for search dynamically after each search iteration based on

1. until a goal is reached
2. perform a best-first search on $\hat{f}$ until *time bound*
3. update heuristic values of nodes in CLOSED
4. $s \leftarrow$ state in OPEN with the lowest $\hat{f}$
5. start executing path to s
6. *time bound* $\leftarrow$ execution time to reach s
7. *OPEN* $\leftarrow \{s\}$; clear CLOSED

Figure 1: Pseudocode for the Dynamic $\hat{f}$ algorithm.

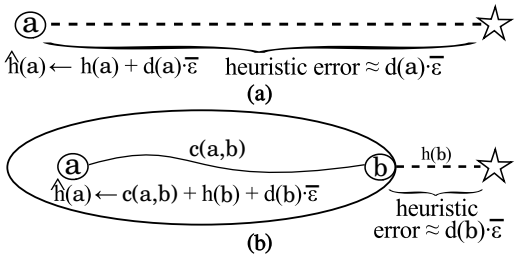


Figure 2: Backed up heuristic error in Dynamic $\hat{f}$.

the duration of the actions that have been queued for execution (line 6). The longer it will take the agent to execute the actions that have been committed to, the more computation can be done by the search before it must terminate the subsequent iteration. This can result in a positive feedback loop of longer queued paths and yet more search time, allowing the search to perform larger learning steps and ultimately reaching the goal much faster. In contrast, LSS-LRTA* tries to minimize total search effort, and thus avoids search during execution. Under the GAT metric, concurrent search can be regarded as free.

The second modification that Dynamic $\hat{f}$ employs is an inadmissible heuristic, notated $\hat{h}$ (line 2). This value is our unbiased best guess about what the node's true f^* value is, rather than a lower bound. Just as $f(s) = g(s) + h(s)$, we will write $\hat{f} = g(s) + \hat{h}(s)$. In Dynamic $\hat{f}$, the search frontier is sorted on $\hat{f}$ instead of f. While any $\hat{h}$ could be used, in experiments report below, we use a version of the admissible h that is debiased online using the 'single-step error global average' method of Thayer, Dionne, and Ruml (2011). During search, the error ϵ in the admissible h is estimated at every expansion by the difference between the f value of the parent node and the f value of its best successor (these will be the same for a perfect h). If $\bar{\epsilon}$ is the average error over all expansions so far and $d(s)$ is an estimate of the remaining search distance (number of edges along the path) from a state s to the nearest goal, then $\hat{h}(s) = h(s) + \bar{\epsilon} \cdot d(s)$. Inadmissible heuristics have shown promise in other suboptimal search settings as well (Thayer 2012).

In Dynamic $\hat{f}$, the next action to take from the current state is the one leading to the successor with the best $\hat{f}$ value. As noted by Burns, Kiesel, and Ruml (2013), this requires some subtlety. If node a inherits its f value from a node b on the search frontier, then the heuristic learning step (line 3) will update its h value using the path cost between a and

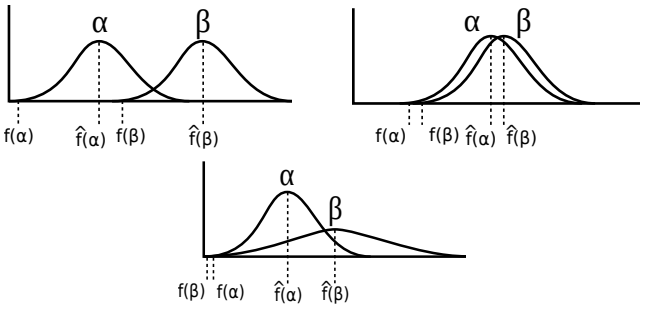


Figure 3: The three scenarios considered by Ms. A*.

b (the difference in their g values) and b's h value. The remaining error in the estimate of a's f value will then derive from the error in b's h value. This is illustrated in Figure 2. To take this into account, we compute $\hat{h}(a) = h(a) + \bar{\epsilon} \cdot d(b)$. This means that, as we learn updated h values, we record the d values of the nodes they were inherited from.

Metareasoning

One of the first applications of metareasoning to heuristic search was Decision-Theoretic A* (DTA*) (Russell and Wefald 1991). DTA* emits actions individually, like a real-time search, but at each step it estimates how much search to perform. To do this, it weighs the time required to search against the likelihood that further search will change the selected action and the expected reduction in total GAT. If further search is predicted to reduce the expected GAT more than the time taken for the searching, then the algorithm implicitly executes an identity action and performs additional search. This approach is extremely elegant (and inspired the approach taken in this paper). However, to estimate the effects of further search, DTA* uses offline training data, which can be laborious to gather. Furthermore, DTA* also makes the assumption of a search space with disjoint subtrees, searching independently under the competing actions that the agent might take next. Burns (2013) showed that it performs poorly compared to modern search algorithms like LSS-LRTA* that use a single lookahead search space.

More recently, a metareasoning approach has been employed for search in the form of regret minimization in MDPs (Tolpin and Shimony 2012). This work attempts to estimate the probabilistic gain of performing additional rollouts in a stochastic environment. Like DTA*, it considers the actions available at the agent's current state, and makes its decision based on the likelihood of additional computation showing that the currently best action is surpassed by a competitor. This is done by continually sampling the top level actions until the incumbent is deemed sufficiently more promising than its competitors.

A simplistic metareasoning analysis is used by the Ms. A* algorithm of Burns (2013) to determine whether to take identity actions and how much of the plan prefix to commit to. Like Dynamic $\hat{f}$, Ms. A* uses an unbiased inadmissible heuristic $\hat{h}$ and defines $\hat{f} = g + \hat{h}$. Define α as the action whose unbiased $\hat{f}$ is lowest, and β as a competing action to be taken. As indicated in Figure 3, for each action, Ms. A* interprets $\hat{f}$ as the expected value of a belief distribution regarding the location of the true f^* cost of the action and interprets the admissible f value as the truncated left edge of that distribution. (This is reminiscent of work on Bayesian reinforcement learning (Dearden, Friedman, and Russell 1998).) By definition, $\hat{f}(\alpha) \leq \hat{f}(\beta)$. If $f(\alpha) \leq f(\beta)$ as well, then Ms. A* concludes that either α is significantly better than β (top left panel in the figure) or the two are so close that further search is not worth the negligible gain (top right panel). However, if $f(\beta) < f(\alpha)$ (lower panel), then Ms. A* concludes that further search is warranted. While this analysis is quick and easy, it completely ignores the cost and likely effects of further search. The simplicity of Ms. A* can lead it to take identity actions when they do not result in improved results. O'Ceallaigh (2014) showed empirically that Ms. A* can sometimes outperform previous real-time algorithms, but that it is itself outperformed by the more principled algorithms presented below.

Deciding When to Act

We now turn to the first of the two metareasoning schemes we propose in this paper. It concerns 'identity actions', which are special actions available in some domains that the agent may take but that don't change the problem state, but merely allow the agent to delay acting. (Such an action is also known as a 'no-op', although we prefer to emphasize that we are assuming the state does not change.) They allow the agent to continue searching from the same initial state as in the previous search iteration, without discarding the lookahead search space, and therefore they allow the agent to look further ahead and gather more information about the potential long-term effects of choosing the different actions applicable in the current state.

We call our algorithm $\hat{f}_{IMR}$, as it decides whether to take an identity action by using a metareasoning process like that of DTA*. It is a variant of Dynamic $\hat{f}$. Whenever an identity action is applicable in the current state, the algorithm attempts to estimate whether the time that could be spent planning during the identity action will be more than offset by the expected improvement in GAT resulting from being more likely to select a better action. If so, the learning step is skipped, an identity action is issued, and the search continues from where it left off.

More formally, if $t_{identity}$ is the duration of the identity action and B is the expected benefit of search (in terms of GAT reduction), then $\hat{f}_{IMR}$ will take the identity action iff

$$B > t_{identity}. \quad (1)$$

The value of $t_{identity}$ is known (likely selected by the system designer); it is B that is more difficult to estimate. Note that search will only provide benefit if, instead of taking the currently most promising action α, we select some other action β instead. So we must estimate the probability that, after searching, our estimate of β's expected cost has fallen below α's, and if so, by how much. Note that we choose actions based on their $\hat{f}$ values, so what we need to know is where α and β's $\hat{f}$ values are likely to be after we have

performed more search. As in any metareasoning approach, we will need to make some significant assumptions and approximations in order to make our method practical.

We approach this problem from a perspective similar to that of Ms. A*: we view our belief about the value of an action a as a probability distribution over possible values, with $p_a(x)$ representing the probability density that a's true f^* value is x. Thinking back to Dynamic $\hat{f}$, we can see that further lookahead will decrease the error in a's backed up f value, represented in our $\hat{f}$ estimate as the d value that a inherits from its best frontier descendant b. If we view our $p_a(x)$ belief distribution, as we did with Ms. A*, as centered on $\hat{f}$ with variance controlled by $\bar{\epsilon} \cdot d(b)$, then we see that additional search will decrease our uncertainty about a's value, as we would expect. More specifically, because $\bar{\epsilon} \cdot d(b)$ represents our estimate of the expected error in a's f value ($\hat{f}(a) - f(a)$), we interpret it as an estimate of the standard deviation of p_a, and its square as the variance of the belief distribution: [1]

$$\sigma^2_{p_a} = (\bar{\epsilon} \cdot d(b))^2. \quad (2)$$

Now what we need in order to compute B is an estimate of where $\hat{f}(a)$ might be located if we were to have performed additional search. We represent this too as a probability distribution, with $p'_{\bar{a}}(x)$ representing the probability density that $\hat{f}(a) = x$ after further search. We model $p'_{\bar{a}}$ as a Gaussian distribution. We first note that, because the current $\hat{f}(a)$ value is our best guess about the true value of a, we can use it as the mean of $p'_{\bar{a}}$. Second, we note that, if no further search were done, the variance of $p'_{\bar{a}}$ should equal that of p_a, and if we searched all the way to a goal, then the variance of $p'_{\bar{a}}$ would be zero, since we would know its true value. Therefore, we make the (admittedly strong) assumption that the variance of $p'_{\bar{a}}$ is

$$\sigma^2_{p'_{\bar{a}}} = \sigma^2_{p_a} \cdot (1 - \min(1, \frac{d_s}{d(b)})) \quad (3)$$

where d_s is the distance, in search steps, along the path to a goal that we expect to cover during the search. In other words, we take the variance of p_a as the variance of $p'_{\bar{a}}$, but scaled according to how far toward the goal we expect to get. To estimate d_s, we use the concept of expansion delay introduced by Dionne, Thayer, and Ruml (2011). Expansion delay estimates the average progress of a search along any single path. It is easily calculated as the average number of expansions from when a node is generated until it is expanded. Our implementation tracks path-based averages and uses the average computed during one iteration as the value for the next. Given the number of expansions that the search will perform within the duration of the candidate identity action,

$$d_s = \frac{expansions}{delay}. \quad (4)$$

This gives us all the ingredients for our belief distribution:

$$p'_{\bar{a}} \sim N(\hat{f}(a), \sigma^2_{p'_{\bar{a}}}). \quad (5)$$

Now that we can estimate how our beliefs about the values of actions might change with search, we can return to our central concern: estimating the possible benefit of search. If the value of the most promising action α were to become x_α and the value of some competing action β were to become x_β, the benefit would be

$$b(x_\alpha, x_\beta) = \begin{cases} 0 & \text{if } x_\alpha \leq x_\beta \\ x_\alpha - x_\beta & \text{otherwise} \end{cases} \quad (6)$$

because we would have done α if we had not searched. Now we just compute the expected value over our estimates of $p'_{\bar{\alpha}}$ and $p'_{\bar{\beta}}$:

$$B = \int_{x_\alpha} p'_{\bar{\alpha}}(x_\alpha) \int_{x_\beta} p'_{\bar{\beta}}(x_\beta) b(x_\alpha, x_\beta) dx_\beta dx_\alpha. \quad (7)$$

In the implementation tested below, we use a straightforward numerical integration with 100 steps.

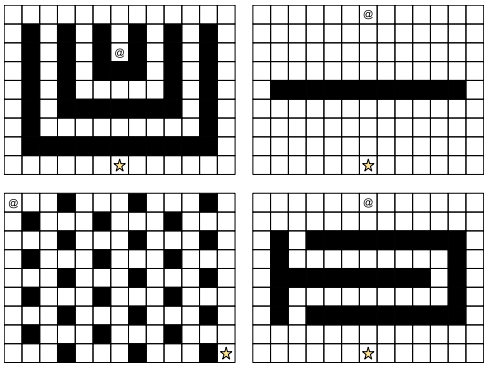


Figure 4: Handcrafted pathfinding problems. Clockwise from top left: nested cups, wall, slalom, uniform.

Experimental Results

To assess whether $\hat{f}_{IMR}$ behaves as we would expect, we constructed four simple handcrafted instances of grid pathfinding, where it is easy to assess algorithm behavior. To gauge whether $\hat{f}_{IMR}$ might be useful in practice, we then tested it on four larger more realistic real-time search benchmark domains.

Handcrafted Instances

Small versions of the four handcrafted pathfinding problems are sketched in Figure 4, with @ marking the start and a star marking the goal. Movement was four-way and the heuristic was Manhattan distance, ignoring obstacles. To see a full spectrum of behaviors and have a good basis for assessment, we tested A*, an off-line optimal search; RTA*, a simple

[1] An implementation detail: although the lookahead search of $\hat{f}_{IMR}$ uses the same 'single-step global average' method for estimating $\bar{\epsilon}$ as Dynamic $\hat{f}$ did, preliminary studies (O'Ceallaigh 2014) showed that the alternative 'path-based correction' approach of Thayer, Dionne, and Ruml (2011) gave a better estimate when used to compute the variance of the desired belief distribution, so that method is used when computing variance.

instance	algorithm	GAT	short	identity
cups	A*	166	90	90
	hill-climbing	3108	1	1
	RTA*	1666	1	1
	LSS-LRTA*	3500	1	1
	$\hat{f}$	5322	1	1
	$\hat{f}_{IMR}$	970	255	255
	$\hat{f}_{PMR}$	4238	646	1
	Mo'RTS	241	83	82
wall	A*	102	43	43
	hill-climbing	241	1	1
	RTA*	723	1	1
	LSS-LRTA*	523	1	1
	$\hat{f}$	717	1	1
	$\hat{f}_{IMR}$	101	31	31
	$\hat{f}_{PMR}$	441	100	1
	Mo'RTS	140	64	62
uniform	A*	29578	27180	27180
	hill-climbing	2999	1	1
	RTA*	2997	1	1
	LSS-LRTA*	3195	1	1
	$\hat{f}$	2997	1	1
	$\hat{f}_{IMR}$	2997	1	1
	$\hat{f}_{PMR}$	2997	1	1
	Mo'RTS	2997	1	1
slalom	A*	177	27	27
	hill-climbing	1974	1	1
	RTA*	2168	1	1
	LSS-LRTA*	382	1	1
	$\hat{f}$	638	1	1
	$\hat{f}_{IMR}$	161	6	6
	$\hat{f}_{PMR}$	4794	662	1
	Mo'RTS	161	6	6

Table 1: Performance on handcrafted pathfinding instances.

real-time search; LSS-LRTA*, a modern real-time search; and Dynamic $\hat{f}$, the algorithm that $\hat{f}_{IMR}$ extends. We also tested a greedy hill-climbing algorithm that performs only one expansion and moves to the best child (equivalent to RTA* with a lookahead of one). In addition to measuring our central figure of merit, GAT, we also counted the number of short trajectories, when an algorithm chooses not to commit to all the actions along the best path to the lookahead frontier, and the number of identity actions executed. The two counts are identical for $\hat{f}_{IMR}$ but will differ for later algorithms in this paper. For all algorithms, the first search iteration is considered an identity action. For A*, all iterations are identity actions until the search is complete and the agent starts moving. To simplify reasoning about the algorithms, we disabled dynamic lookahead, so Dynamic $\hat{f}$ and $\hat{f}_{IMR}$ act like LSS-LRTA* and wait to search until the last committed action has begun executing. The lookahead (or equivalently, the time per action) was set to 10 expansions.

The first instance, nested cups, has sets of walls forming nested local minima which temporarily ensnare real-time searches. The full instance was 51×29 ($w \times h$) with corridors two spaces wide and a 3×3 space in the innermost cup. One would expect A* to outperform hill-climbing, for example, because A* expands each state at most once, while hill-climbing and similar real-time searches must revisit the same states repeatedly as they build up enough learning to escape the minima. Experimental results are shown in Table 1, and indeed A* has the best performance, while most of the real-time methods suffer (the $\hat{f}_{PMR}$ and Mo'RTS algorithms will be introduced below). Notably, however, $\hat{f}_{IMR}$ is able to detect that the heuristic is deceptive and that planning ahead is useful, as it executes a substantial number of identity actions and performs 5.5 times better than the $\hat{f}$ method it is based on. We had expected $\hat{f}$ to perform well, but as it learns that the heuristic significantly underestimates, it increases $\bar{\epsilon}$, which causes it to behave more greedily. With an unreliable heuristic, $\hat{f}_{IMR}$ chooses more deliberation, which appears to be a better strategy.

The wall instance, where a single flat obstacle in the middle of the map blocks the goal, elicits similar behavior because the wall creates a single large local minimum. The full instance was 41×21 with a gap of one on either side of the wall. $\hat{f}_{IMR}$ executes many identity action and performs just as well as A*, while $\hat{f}$ is 7 times worse.

In the uniform instance, where small obstacles are uniformly distributed across the map, the local minima each require only one step to escape. The full instance was 1200×1200. We would expect the real-time algorithms to perform well, while A* would labor to determine the true optimal path among many close contenders. The results confirm these expectations, with $\hat{f}$ reaching the goal in a tenth of the time of A*. $\hat{f}_{IMR}$ recognizes that identity actions are not needed and matches the performance of the other real-time searches.

The final handcrafted instance, slalom, features a long and winding path down the center of the map to the goal, with a quicker option being to bypass the slalom via either side of the map. The full instance was 37×124 with the corridor 2 spaces wide and a 3-space margin around the outside. As the results indicate, the simple real-time algorithms commit to following the winding path while A* is able to find the outside path and reach the goal much faster. LSS-LRTA* and $\hat{f}$ start down the path but eventually turn back, while $\hat{f}_{IMR}$ quickly recognizes the deceptive heuristic, executes a few key identity actions, and reaches the goal even faster than A*.

To summarize, $\hat{f}_{IMR}$ appears able to successfully adapt to behave more like A* or more like hill-climbing as the situation requires. While these small benchmarks are very promising, it remains to be seen if the algorithm can perform well on full-scale benchmarks.

Larger Benchmarks

We used four benchmarks: grid pathfinding in the orz100d map from Dragon Age: Origins using the 25 start/goal combinations for which the optimal solution cost was high-

est (Sturtevant 2012), Korf's 100 instances of the 15-puzzle (Korf 1985), 100 randomly selected instances of a platformer-style video game (Burns, Kiesel, and Ruml 2013), and 25 randomly selected instances of a Frogger-style traffic avoidance game (O'Ceallaigh 2014). The video games are examples of dynamic domains closer to robotics, where users expect agents to begin acting promptly and achieve goals quickly, while the sliding tile puzzle is a classic benchmark. While these domains are deterministic and fully observable, the time pressure of optimizing GAT highlights the trade-off between deliberation and acting. All but the traffic domain feature identity actions.

We used full dynamic lookahead for Dynamic $\hat{f}$ and $\hat{f}_{IMR}$. We tested at five different 'search speeds', varying the number of expansions allowed per unit of action duration in powers of 10 from 10^2 to 10^6 inclusive. For each instance, algorithms were given a limit of 7 GB of memory and 10 minutes of CPU time. No data point is plotted for any setting where an algorithm failed to reach the goal on one or more instances within the resource constraints.

We also tested RTA* and LRTA* (Korf 1990), DTA*, and Ms. A*, but their results were inferior and uninteresting, so we do not include them in the plots.

The left column of Figure 5 presents the results of $\hat{f}_{IMR}$, with each row showing a different domain. The x axis of each plot varies the search speed and the y axis shows the GAT, normalized as a factor of the GAT of an oracle that immediately commits to an optimal plan without searching, shown on a $\log_{10}$ scale. The normalization reduces the variance within each domain, as some instances are easier than others. Error bars show 95% confidence intervals on the mean over all the instances in the domain.

The top left panel shows the pathfinding domain. The heuristic is quite accurate and A* is hard to beat. But among the real-time algorithms, $\hat{f}_{IMR}$ shows a slight advantage when few expansions can be performed. The second panel shows the 15-puzzle, and $\hat{f}_{IMR}$ gives a pronounced advantage. The trade-off between behaving like A* when search is fast and behaving like Dynamic $\hat{f}$ when search is slow is evident. In the platformer domain, Dynamic $\hat{f}$ and $\hat{f}_{IMR}$ perform similarly, both better than LSS-LRTA*. And in the traffic domain, there are no identity actions, so $\hat{f}_{IMR}$ is identical to Dynamic $\hat{f}$, both of which are better than LSS-LRTA*.

In summary, results in both the small handcrafted instances and the larger benchmarks suggest that a metareasoning approach to deciding when to commit to actions and when to plan further is quite promising, matching or outperforming existing real-time algorithms.

Deciding How Many Actions to Commit To

We now turn to the second of the two metareasoning schemes we propose in this paper. Recall that modern real-time search algorithms like LSS-LRTA* and Dynamic $\hat{f}$ explore a local search space and then commit to a plan prefix leading to a frontier node (line 5 in Figure 1). While $\hat{f}_{IMR}$ addressed the issue of whether to commit or search further, the issue we consider now is, given that we commit to acting, how much of the plan prefix should we commit to? If there exists some state s along the prefix P where the action α_s selected at s is not certainly better than an alternative β_s, it might prove worthwhile to cut P short such that it ends at s. This path prefixing operation allows the search to pay attention to promising paths which might otherwise be ignored. We call this algorithm $\hat{f}_{PMR}$, as it considers prefixes using metareasoning.

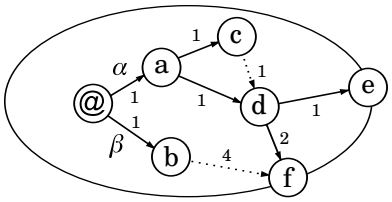


Figure 6: Useful (d) versus not useful (@, a) decision points.

We use the same assessment of the benefit of search as in $\hat{f}_{IMR}$, computing B at the nodes along P and stopping at the first one for which search appears worthwhile. In this situation, we are not comparing the benefit B against the duration of an identity action, but rather against the more nebulous costs of 'starting search at a point before the frontier'. We tested two approaches to assessing these costs. The first was to assume that they were zero, leading us to commit to search at the first node for which B was positive. The second, and more successful, was to interpret the cost of stopping short of the frontier as the expected time required to regenerate the nodes from the new start state to the current lookahead frontier. We estimate this as the number of steps in the pruned suffix of P times the expansion delay, divided by the number of expansions per action duration. While this does estimate the amount of repeated work, it is not fully satisfactory, as this repeated work will likely be done concurrently with search and might not lead to increased GAT. However, it will certainly prevent the search from looking as far ahead in the search space as it would if we committed to the entire prefix, because a smaller prefix will have a smaller duration (line 6 in Figure 1).

A second complication is that we only wish to consider performing more search at nodes along P that have successors that lie on best paths to different frontier nodes. While there will likely be alternative actions available at every step of the path, some of them may represent unnecessarily expensive temporary deviations from P, rather than truly useful alternatives. Figure 6 gives a concrete example. Edges drawn with solid lines also represent parent pointers back from the successor. The best path P is $\langle$@, a, d, e$\rangle$. Node d represents a useful decision point because it has multiple successors that lie along best paths to different frontier nodes. Node a is not a useful decision point, because c merely represents a longer way to reach the same frontier node as d. The agent's current state is also not a useful decision point, as b does not lie along the best known path to f, so there is no use in going that way. Computing usefulness just requires bookkeeping during the learning step, record-

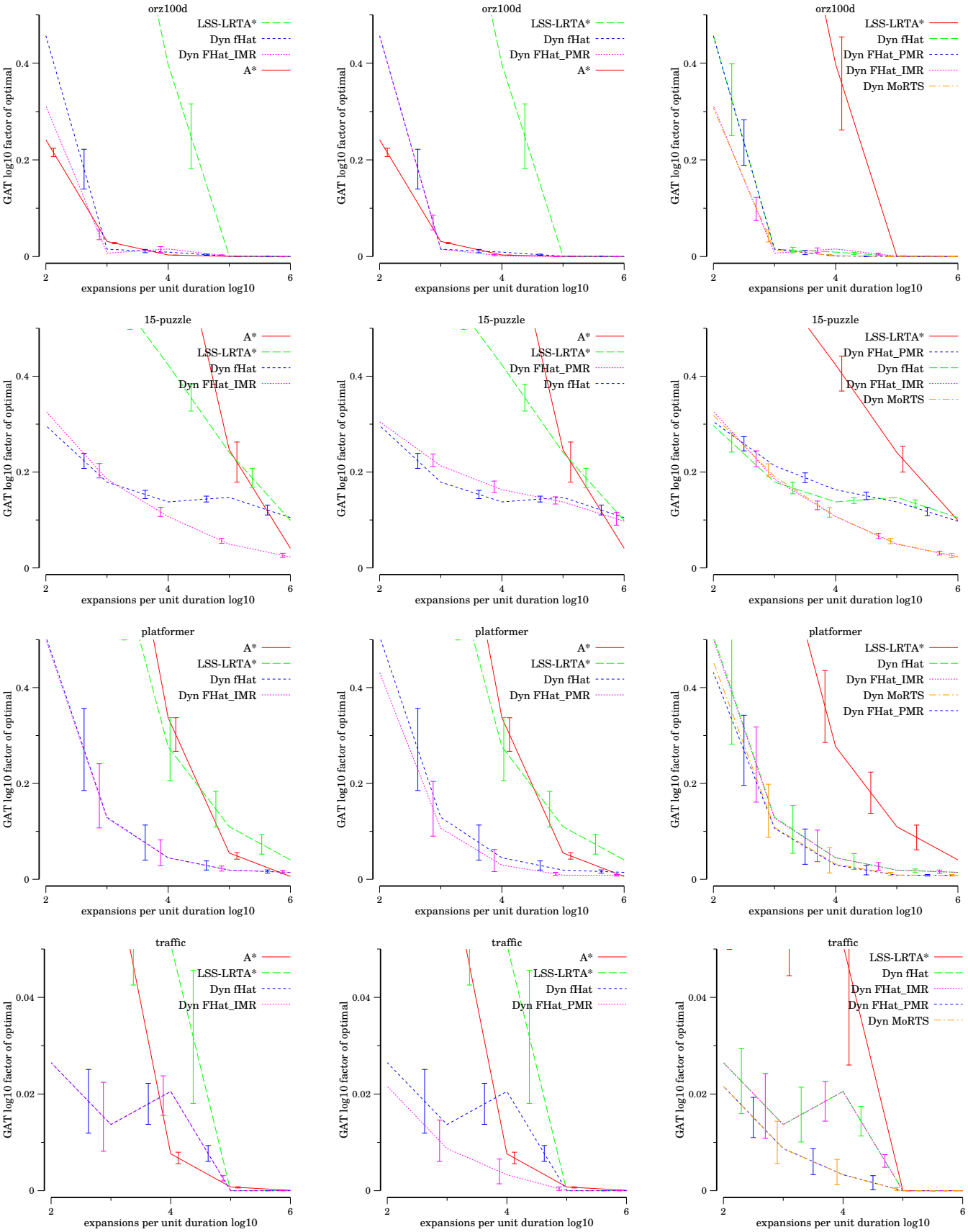


Figure 5: Goal achievement time as a function of search speed. One problem domain per row, different algorithms per column.

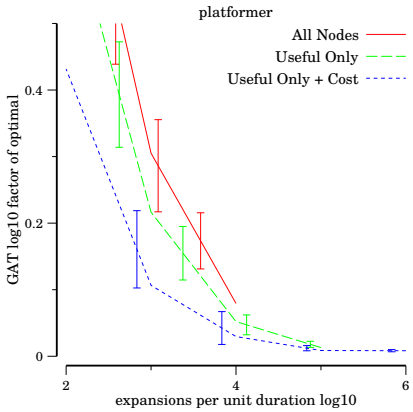


Figure 7: Variants of $\hat{f}_{PMR}$.

ing for each node the frontier node from which it inherits its value, or nil if the inheritance happened through a non-parent pointer (indicating that the node does not lie on a best path to the frontier). $\hat{f}_{PMR}$ then only performs its metareasoning at useful decision nodes along P, where at least two successors lie on best paths to distinct non-nil frontier nodes.

Experimental Results

Figure 7 presents an example of how considering only nodes representing useful choices and considering even a rough cost for pruning a plan prefix leads to improved performance for $\hat{f}_{PMR}$.

The performance of $\hat{f}_{PMR}$ on the handcrafted instances is included in Table 1. While it led to some improvements over $\hat{f}$ on cups and wall, it performed poorly on slalom. Overall, it appears to provide less of a benefit than $\hat{f}_{IMR}$. Its behavior on the larger benchmarks is shown in the middle column of Figure 5. It performs comparably to Dynamic $\hat{f}$ except on traffic, where it outperforms all other algorithms. The performance on larger benchmarks is reminiscent of $\hat{f}_{IMR}$, in that $\hat{f}_{PMR}$ performs similarly to Dynamic $\hat{f}$ except on one domain, where it shows a pronounced advantage. This brings up the obvious possibility of combining the two methods.

Mo'RTS

We investigated the combination of both the identity and prefix techniques in the same algorithm, which we call Mo'RTS (for metareasoning online real-time search, pronounced 'Moe RTS'). Mo'RTS checks if an identity action is applicable at the current state even if it is not a true decision node. Only if acting seems preferable to search is the best path checked for the prefix length to which the algorithm should commit.

The performance of Mo'RTS on the handcrafted instances is included in Table 1. Surprisingly, it outperforms both $\hat{f}_{IMR}$ and $\hat{f}_{PMR}$ on the cups instance, coming very close to A*'s performance. On the wall instance, it performs almost as well as $\hat{f}_{IMR}$, and the same as $\hat{f}_{IMR}$ on uniform and slalom. Performance on larger benchmarks is shown in the right column of Figure 5. On pathfinding and the 15-puzzle, where $\hat{f}_{IMR}$ is strong, Mo'RTS does just as well. And on traffic and platformer, where $\hat{f}_{PMR}$ is strong, Mo'RTS matches its performance. Overall, the results show significant improvement over the state-of-the-art LSS-LRTA* and Dynamic $\hat{f}$ algorithms.

Discussion

While the methods we introduce appear to work well across a variety of domains, they are based on several assumptions. First, both Dynamic $\hat{f}$ and metareasoning methods assume access to an inadmissible $\hat{h}$, which in this work we create using on-line debiasing. There is no strong practical theory that we currently know of to explain when such a method will result in a reasonable heuristic or not. Thayer (2012, Figure 4-3) and O'Ceallaigh (2014, Table 3.2) suggest that debiasing yields an inaccurate heuristic, yet it is clearly effective. Second, we make several assumptions in order to approximate $p'_{\bar{a}}$, including a Gaussian form, a linear variance reduction with lookahead, and a prediction of future expansion delay. In our limited investigations (O'Ceallaigh 2014), our Gaussian approximation of $p'_{\bar{a}}$ seems remarkably poor, and it is surprising that the metareasoning algorithms work as well as they do. There is likely much room remaining for improvements.

As we mentioned above, we do not currently have a fully satisfactory way in $\hat{f}_{PMR}$ to understand the implicit cost of choosing to commit to a plan prefix that stops short of the lookahead frontier. Using a short prefix results in less search time for the following iteration, which limits the number of actions to which that iteration may commit. Both $\hat{f}_{IMR}$ and $\hat{f}_{PMR}$ are essentially myopic. Explicitly reasoning about all this may be too expensive for on-line metareasoning.

In this study, we limited lookahead in node generations, not wall time. While this simplifies replication of results, it ignores the many issues necessary for deploying real-time search, such as predictable OS interrupt servicing and memory management. Considering an identity action is only done once per search iteration, if one is applicable at the agent's current state, and so the CPU overhead is likely negligible. Consider possible prefixes is done at potentially every node of the best path to the frontier, if all nodes are useful. (Computing usefulness adds negligible overhead, as explained above.) It remains to be seen whether this overhead is significant in practice, although in most domains there are many times more nodes generated than there would be along any single path to the frontier.

If the overhead of metareasoning could be made low enough, it may be beneficial to check more frequently whether the current lookahead gives sufficient confidence for committing to one or more actions. This would decouple the search iterations from the action start/end times. It would also provide an alternative approach to considering path prefix decisions.

Our study has demonstrated concurrent planning and execution with a real-time state-space planner. The real-time

search approach certainly applies to a plan-space planner, but the agent commits to decisions in the order in which they are made, which may be awkward if decisions early in the search space concern actions that cannot be executed until later. In this sense, real-time search encourages the search space to place decisions that the agent is currently facing early in the search space.

Metareasoning has previously been used for directly guiding expansion decisions in off-line search, in which all planning occurs before any acting (Burns, Ruml, and Do 2013), and in contract search, where the planner faces a deadline (Dionne, Thayer, and Ruml 2011). It has also been used to decide which of multiple available heuristics to use in A* (Tolpin et al. 2013), IDA* (Tolpin et al. 2014), and CSP solving (Tolpin and Shimony 2011). This recent generation of work is fulfilling the early promise heralded by pioneers from the late 1980s such as Dean and Boddy (1988) and Russell and Wefald (1991).

In the context of robotics, our study has illustrated concurrent planning and acting, but only on deterministic domains. However, because each iteration of a real-time search can consider an updated world model or even updated goals, the approach should generalize easily to settings in which new information or new goals arrive, or in which execution failures occur.

Conclusion

In this work, we considered how metareasoning can be applied to the problem of concurrent planning and acting. We presented two methods for deciding how to commit to actions during a real-time search and investigated their behavior on both small easily-understood benchmarks and larger more realistic problems. Our techniques allow a single algorithm to dynamically adapt its behavior to the problem at hand, quickly committing to actions like greedy hill-climbing when possible or deliberating before acting like A* when necessary. Empirically, our methods match or outperform the state-of-the-art Dynamic $\hat{f}$ real-time search algorithm. This work provides a principled perspective on current planning and acting, and we hope to generalize it to address additional challenges in planning for robotics.

Acknowledgments

We gratefully acknowledge support from NSF (award 1150068), preliminary work by Sofia Lemons, code by Scott Kiesel and Ethan Burns, and discussions with Solomon Shimony, David Tolpin, and Ariel Felner.

References

Burns, E.; Kiesel, S.; and Ruml, W. 2013. Experimental real-time heuristic search results in a video game. In *Proceedings of the Sixth International Symposium on Combinatorial Search (SoCS-13)*.

Burns, E.; Ruml, W.; and Do, M. B. 2013. Heuristic search when time matters. *Journal of Artificial Intelligence Research* 47:697–740.

Burns, E. 2013. *Planning Under Time Pressure*. Ph.D. Dissertation, University of New Hampshire.

Dean, T. L., and Boddy, M. S. 1988. An analysis of time-dependent planning. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI-88)*, 49–54.

Dearden, R.; Friedman, N.; and Russell, S. 1998. Bayesian Q-learning. In *Proceedings of AAAI-98*, 761–768.

Dionne, A. J.; Thayer, J. T.; and Ruml, W. 2011. Deadline-aware search using on-line measures of behavior. In *Proceedings of the Symposium on Combinatorial Search (SoCS-11)*. AAAI Press.

Ghallab, M.; Nau, D.; and Traverso, P. 2014. The actor's view of automated planning and acting: A position paper. *Artificial Intelligence* 208:1–17.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics* SSC-4(2):100–107.

Hernández, C.; Baier, J.; Uras, T.; and Koenig, S. 2012. Time-bounded adaptive A*. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS-12)*, 997–1006.

Koenig, S., and Sun, X. 2009. Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems* 18(3):313–341.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence* 42:189–211.

O'Ceallaigh, D. 2014. Metareasoning in real-time heuristic search. Master's thesis, University of New Hampshire.

Russell, S., and Wefald, E. 1991. *Do the Right Thing: Studies in Limited Rationality*. MIT Press.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144–148.

Thayer, J. T.; Dionne, A.; and Ruml, W. 2011. Learning inadmissible heuristics during search. In *Proceedings of the Twenty-first International Conference on Automated Planning and Scheduling (ICAPS-11)*.

Thayer, J. 2012. *Heuristic Search Under Time and Quality Bounds*. Ph.D. Dissertation, University of New Hampshire.

Tolpin, D., and Shimony, S. E. 2011. Rational deployment of CSP heuristics. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, 680–686.

Tolpin, D., and Shimony, S. E. 2012. MCTS based on simple regret. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Tolpin, D.; Beja, T.; Shimony, S. E.; Felner, A.; and Karpas, E. 2013. Toward rational deployment of multiple heuristics in A*. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-13)*.

Tolpin, D.; Betzalel, O.; Felner, A.; and Shimony, S. E. 2014. Rational deployment of multiple heuristics in IDA.

In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI-14)*, 1107–1108.

Robust Efficient Robot Planning through Varying Model Fidelity

Breelyn Kane Styler and **Reid Simmons**
Robotics Institute
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
{breelynk,reids}@cs.cmu.edu

Abstract

Long-term robot autonomy requires reliable execution success. Execution success improves by modeling the real world accurately during planning time. Unfortunately, computing a full plan in highly accurate models is often intractable since accurate world models increase the dimensionality of the planning space. Additionally, planning in the highest model is unnecessary for simple uncluttered parts of the environment. Current planners achieve tractable planning times by using approximate models. However, these approximations reduce accuracy resulting in decreased execution success. We introduce an approach that balances planning time efficiency while maintaining execution success. This is accomplished by leveraging a varying fidelity model hierarchy. The approach identifies infeasible execution locations and uses a model selection process to locally re-plan in the minimum fidelity model necessary to circumvent the infeasibility. This effectively generates a single, mixed-fidelity plan. We evaluate the approach on a simulated differential drive robot that navigates a constrained environment. The robot maintains its rate of successful task completion while conserving computation resources by switching from lower fidelity models only when needed.

1 Introduction

Autonomous robots leverage planning and models to intelligently operate in the real world. Furthermore, autonomous robots deployed for long periods need to be robust to failure. They can minimize runtime failures by reasoning about models that capture execution feasibility at plan-time. Models can be arbitrarily complex to better represent the interaction between the robot and its environment. These high fidelity models often include higher dimensions, dynamics and differential constraints. However, planning with such complex models is computationally expensive. Therefore, model approximation may be necessary to achieve tractable global planning times for complex tasks. The loss of model fidelity from these approximations leads to increased failure rates in complex locales of the environment where lower fidelity models are insufficient.

Model approximations sacrifice fidelity for computation feasibility. These approximations are sufficient for simple regions in the task space. For example, a model that considers detailed terrain vehicle interactions is unnecessary for large flat open spaces. They are insufficient, however, for complex portions of the environment. For that reason, the robot's non-uniform execution space suggests plan savings by leveraging multiple models. Our work attempts to stay as long as possible in the lowest fidelity model applicable in order to decrease plan computation time without sacrificing execution quality.

Our approach organizes a given set of planning models into a directed hierarchical graph. These varying fidelity models, when combined, approximate the continuous highest possible real-world model. The approach attempts to detect the parts of the lower fidelity plans that are infeasible for execution and repair them using re-planning through higher fidelity model selection. The model selection process uses prior plans to autonomously select the most applicable higher fidelity model in the hierarchy. This higher fidelity model is used to locally plan to an intermediate goal where the previous lower fidelity plan is resumed. This approach creates a mixed model plan.

Our approach increases robot robustness by giving robots the ability to autonomously decide to re-plan in higher fidelity model representations that inherently incorporate more information. This will improve performance over only using a lower fidelity model, and it will reduce planning times over only using the highest fidelity model. Model switching combines the robustness of a high-fidelity plan with the efficiency of an abstract representation by using higher fidelity models only when necessary.

We ran a series of experiments, in simulation, with a differential drive robot, and used wheeled mobile robot motion planning models for our testing. In our results, we demonstrate failure rates and planning times when using a fixed single model versus autonomously switching between models of varying fidelity. Our approach creates plans that maintain robustness, and take significantly less time to plan, overall, than planning from the start using the highest fidelity model.

2 Related Work

Model generation is prevalent in the robotics community. For instance, there are models for humanoid balancing (Stephens 2007), physics based models for manipulation planning (Dogar et al. 2012), and models for wheeled robots that more accurately capture real world trajectories (Seeg-

miller and Kelly 2014). These previous works demonstrate different model types that increase robustness as the fidelity increases. These works inspire the use of a multi-model approach, but they use hard coded rules for knowing what model to use, and do not include a switching strategy.

Adjusting the resolution in a plan could be considered a form of model fidelity switching. The following works adapt the planning space resolution locally around the robot, but do not vary the model fidelity throughout the same plan (Kambhampati and Davis 1986), (Steffens, Nieuwenhuisen, and Behnke 2010), (Behnke 2004). Our strategy varies models throughout the same plan and changes the resolution of both the state and action spaces.

Our use of hierarchical models is inspired by other hierarchical approaches (Fernández-Madrigal and González 2002), (Barbehenn and Hutchinson 1995) as well as hierarchical approaches in task motion planning that recognize the need to consider lower-level motion plans in task space for plan success (Wolfe, Marthi, and Russell 2010), (Kaelbling and Lozano-Pérez 2011).

In addition to a model hierarchy our work uses re-planning to switch between varying fidelity models. This is motivated by previous work in re-planning. Re-planning is a technique that has been in symbolic planning since early mobile robots (Fikes, Hart, and Nilsson 1972). The execution monitoring portion of the planner PLANEX included ways to regenerate unsuccessful plan actions with different arguments. Re-planning is also prevalent in the motion planning community. Motion planning uses re-planning in incremental search (Stentz 1995) (Koenig and Likhachev 2002), and sampling based algorithms (Bruce and Veloso 2002) (Ferguson, Kalra, and Stentz 2006). Our approach re-plans between different hierarchical levels of the configuration space and workspace including reasoning about differential constraints

Many previous works focus on when to switch between models rather than reasoning about what detail the model contains before switching. Our work contains an additional model selection stage that most related works do not. This stage determines what detail level is most applicable for re-planning.

Related work, such as (Howard and others 2009), uses fixed strategies by focusing on when to switch between levels of detail. This is apparent in work that only has two levels, such as a higher level global plan that guides a low level continuous planner (Knepper and Mason 2011). Similarly hybrid planning switches between a distinct discrete plan and more focused continuous planner (Plaku, Kavraki, and Vardi 2010) as occurs in the SyClop planning framework. This also occurs in (Choi, Zhu, and Latombe 1989) where a contingency "channel" with many possible motion plans guides a lower-level potential field controller. Other work (Göbelbecker, Gretton, and Dearden 2011) divides the planning space between task and observation level plans similar to the division between global and local planners. They switch between a fast sequential "classical" planner, that generates an overall strategy, and more expensive decision-theoretic planning for abstracted sub problems.

Multi-modal and multi-stage work also contain fixed switching strategies. They switch between different planner types based on the modal discretization of the motion planning domain. Multi-modal motion planning for humanoid manipulation (Hauser, Ng-Thow-Hing, and Gonzalez-Baños 2011) generate trajectories within a single mode and reason about the motion transitions as targets. The levels of mode "classes" are pre-defined with such modes as: walking, reaching left, reaching right etc, each with their own constraints and dynamics. Work by (Hauser and Latombe 2009) also divides the space into modes based on the different dimensions in motion configuration space. They sample transitive connections between modes but do not reason explicitly about the detail they are switching to. Lastly, work by (Sucan and Kavraki 2011) for task motion multigraphs also contain predefined points where switching occurs. This work decides between left and right arm motion trajectories which could be viewed as different models. The decision of when to switch is predefined between tasks and the selection criteria is based on computation time. Our model selection criteria reasons about task success and the point when to switch to a new model is not predefined.

A work that is more similar to ours graduates motion primitive fidelity along a state lattice (Pivtoraiko and Kelly 2008). They change the fidelity between replans in the motion planning workspace. They also recognize that "partially or completely unknown" regions of the space can use lower fidelity representations than regions most relevant to the current problem. The work also claims that previous multi-resolution work is more systematic while theirs allows different resolution regions to move over time. The amount of fidelity around the robot is fixed and moves like a sliding window, which is different than our mixed fidelity plan.

Our work is mainly inspired by Gochev's previous work with adaptive dimensionality (Gochev, Safonova, and Likhachev 2013) (Gochev et al. 2011) (Gochev, Safonova, and Likhachev 2012). That work divides the state space into two parts; a high dimensional and low dimensional graph with defined transition probabilities. They use a state lattice planner with pre-computed grid transitions. Unlike other works, they are able to mix the two subspaces into a single plan. Our work also has this same effect, of generating plans that mix dimension spaces, through adaptive switching. They also have a tracking phase (similar to our plan checking stage) to determine where to insert higher fidelity states in a low fidelity plan. Our novelty is we use a complete hierarchy of models. Therefore, our algorithm contains an additional model selection stage. In addition, our models incorporate more than just the state space dimensionality.

3 Approach

Our algorithm for robust plan generation consists of three main stages that loop:

1. Plan generation.
   Here a plan is generated for a particular start and goal state using a given model.
2. Feasibility detection.
   This is the plan checking stage where execution problems are determined. This stage answers the question of 'when'

to switch between models in the plan. [1]

3. Model selection.
   This stage decides what model to switch to for re-planning. The model selector reasons over the model graph to determine what model may be sufficient to repair the plan.

Using the motion planning domain as an example, the robot's task is to navigate from a start state to a goal state. Our system uses an environment map to generate a plan from start to goal in the lowest applicable fidelity model (Plan generation). This plan is the initial global plan.

The plan is then checked in the highest fidelity model to determine parts of the plan that might need repairing (Feasibility detection). Even though the high fidelity model is complex, checking a single plan does not incur much computation time.

If re-planning is necessary, due to a detected impediment, the algorithm moves to stage three (Model selection). The model selector finds the lowest fidelity model that can still feasibly generate a repaired plan. This gives further computation savings by using the lowest fidelity model applicable rather than always switching back to the highest available model. The re-planning model is selected by re-testing the partial plan segment, which needs repair, in higher fidelity models. The first model in which checking of the previous plan segment is *unsuccessful* is the model selected for re-planning. This model is assumed to be a more informative approximation of the space.

The algorithm then cycles back to stage one. A new plan is generated in the selected model. The use of the selected model is localized around the detected impediment by re-planning to intermediate goals on the remainder of the infeasible plan rather than re-planning all the way to the original goal. This creates a partial plan. The new partial plan is then merged back into the global plan. The process repeats until the feasibility detection stage does not find any more impediments. If this is the case, the full global plan is determined to be executable.

The next sections describes our definition of model as well as how the varying fidelity models are organized in a directed hierarchical graph. Section 3.3 provides more details about the robust plan generation algorithm.

3.1 Models

Note that the model space is separate from the underlying robot controller. To illustrate this separation, imagine a person racing a car on switchbacks. They are able to apply a controlled skid to take tight turns at high speeds by estimating an internal momentum model. The internal model they use for their driving plan is separate from the underlying application of braking, hitting the gas pedal, and steering that is necessary for controlling the car. Similarly the robot's models provide decision making options through the generation of possible plans. The plan then needs to be translated into control inputs for the robot to execute [2] (see section 4.1).

When we use the word *model* we describe a three-tuple $\{S_r, S_e, A\}$ consisting of:

- S_r : set of robot states $\{s_0 \dots s_\infty\}$
  Example robot states: position, orientation, mass, wheel slip constraints, etc.
- S_e : set of environment states $\{s_0 \dots s_\infty\}$
  Example environment states: position and orientation of obstacles, 3D/2D representation, static or dynamic obstacles, etc.
- A : set of actions $\{a_0 \dots a_\infty\}$
  Example action set: planar motions, equations of motion that include control inputs and differential constraints, motion primitives, or other trajectory generation methods, etc.

Each model includes states and actions that allow the generation of a plan within that model space. The action space varies from geometric spatial actions to differential equations of motion that include input controls. The environment representation (2D vs 3D) also varies through the state space. Where the models come from is not relevant to this paper. We assume they are all provided to us (see, for instance, (LaValle 2006)).

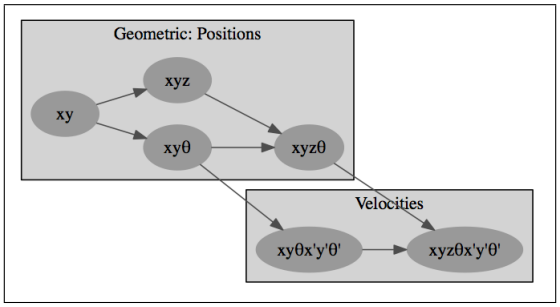


Figure 1: Chassis Directed Model Graph: The graph evolves from (roughly) left to right, with the root at [x,y]. Note: States listed do not encompass the entire model and exist only as labels for the model nodes.

3.2 Model Organization

Ordering the models hierarchically is important for model selection. For a model to be higher fidelity than another, it could be a direct superset by including more states in the space (higher dimensions), or containing more control inputs. The more correct a model is at representing the real

[1] For an execution time-only approach the robot can trigger a failure detection during runtime. This makes sense if failures are non-detrimental and easily detectable. For this work we focus on a plan-time approach since we are testing in simulation and not reasoning explicitly about uncertainty.

[2] Translation is only necessary for the geometric planar models for our specific plan following controller. This translation step can be avoided by planning with directly executable feasible motion primitives, as was demonstrated in previous state lattice motion planning work (Pivtoraiko and Kelly 2005). Additionally, some plan following controllers are robust enough to follow geometric plans without translation. This is demonstrated in Big Dog (Raibert et al. 2008) where the robot successfully executes plans that were generated on a two-dimensional grid.

world (such as modeling slip), or the larger the space considered, the higher fidelity the model is.

Our work makes an assumption that lower fidelity models can be translated into higher models at each layer. This allows the lowest model to be translated into the highest model through each layer of the graph (see Section 4.1). This translation capability is necessary for 1) checking the full plan for errors, 2) model selection, and 3) connecting partial plans with intermediate goals.

As a concrete example, Figure 1 shows a model graph with models from the motion planning community for a two-wheel differential drive robot. The node names are just indicative - the actual models incorporate more information than just a description of state.

As the models change fidelity the environment and/or the action space is changed. For example, the added z-dimension changes only the collision checker to consider three dimensional objects, an environment change. This is because the chassis cannot actuate in the z-dimension, so the addition does not effect the action space. Adding the θ dimension changes both the state and action space. Collision checks are now done in θ and the underlying controller is constrained to s-curve motions. The specific models used in the experiments are described in more detail in Section 4.1.

3.3 Robust Plan Generation

We start by planning in a default model space, typically [x,y], and check the plan in the highest fidelity model. If the plan is feasible, it is sent to the robot for execution. If the plan is not feasible, the infeasible plan segment is sent to the model selector. The model space used for the plan segment that needs repair (initially the [x,y] model) is then the first node to search from in the model graph.

Algorithm 1 Robust Plan Generation

```
1: setupPlanSpace()
2: [p , planResult] = generatePlan(m);
3: globalPlan = SAVEPLAN(p, globalPlan);
4: if planResult == success then
5:     tm = translateToModel(p, highest);
6:     [planCheck, b4repair, afterrepair] = propagateWhileValid(tm, highest)
7:     if planCheck == infeasible then
8:         m = MODELSELECTOR(globalPlan, b4repair, afterrepair);
9:         Goto line 1
10:    else
11:        executeResult = sendToRobot(globalPlan);
12:        if executeResult == failure then Robot failed in execution.
13:        else
14:            Robot made it to goal!
15:        end if
16:    end if
17: else
18:    Failed to find plan.
19: end if
```

Our model selection process uses Breadth First Search to explore the model graph. For each model, we first test the infeasible plan segment in that new model space. If the testing is unsuccessful, it means the model has information not present in the original model used to generate the plan. Replanning from the segment start to intermediate goals (waypoints) is then performed in that model. If a successful plan is found, it is merged back into the global plan (Algorithm 4) to be re-checked. This strategy is detailed in Algorithm 1.

As an example of how the model selector works (Algorithm 2), assume that it starts with the first child of the [x,y] model which is the [x,y,θ] model. The previous [x,y] plan segment is tested in this model (Algorithm 2, line 8). If the plan succeeds, we assume that the [x,y,θ] model does not accurately capture the infeasibility. We then choose the next child of [x,y], which is [x,y,z], and again test the previous plan in this higher fidelity model. If the plan again succeeds, we then try the [x,y,z,θ] model. If testing the previous plan finally does not succeed, we assume this model captures the space of the impediment and we select it as the model to use for re-planning from the start of the infeasible segment. Based on this approach, it is possible to produce a final plan that can effectively skip between model tree levels when choosing the next model (in this example, we skipped from [x,y] to [x,y,z,θ]).

Algorithm 2 The model selector does a Breadth First Search by testing old infeasible plan segments in higher fidelity model spaces until the old plan fails. If the old plan fails, this indicates the model contains information that may be relevant to the impediment and that is the model chosen for re-planning.

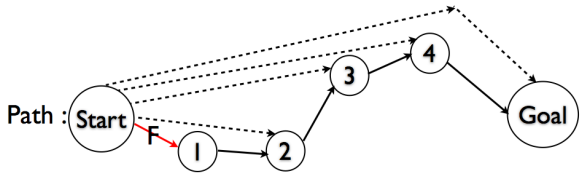
```
1: function MODELSELECTOR(p, b4failIndex, afterfailIndex)
2:     mLast = findModelFor(p.getNode(b4failIndex));
3:     m = mLast;
4:     p = resizePlan(p.getNode(b4failIndex), p.getNode(afterfailIndex));
5:     setUsedModel(m);                    ▷ Last model used becomes root node.
6:     m = getNewModelBFS();
7:     tm = translateToModel(p,m);
8:     planResult = propagateWhileValid(tm, m);        ▷ Does collision checking.
9:     if planResult == success then
10:        Goto line 5
11:    end if
12:    return m
13: end function
```

Intermediate Goals Instead of re-planning in a model all the way to the original goal, the previous plan is potentially reused by planning to intermediate goals. In particular, the remaining nodes after the infeasible area are translated to the currently selected model and set as possible goals. This effectively creates a goal set to plan to (Algorithm 3). If we successfully plan to an intermediate goal we can switch back to using the original plan for the remainder of the plan.

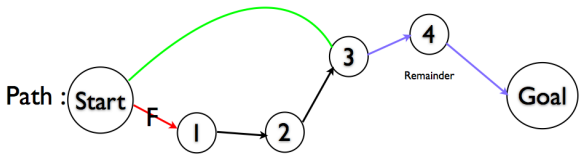
The idea of using waypoints as a cache was also done in work by (Bruce and Veloso 2002). We expand this for multi-fidelity nodes and plan reuse. For instance, in Figure 2 a goal set is created for all unachieved waypoints (1 through goal). The planner then re-plans to this goal set and finds waypoint 3 is successful. The remaining plan, with this new partial plan, is sent for the robot to execute.

Planning to an intermediate goal localizes around the infeasible location in order to plan in higher fidelity models

for less time. The resulting plan to be executed can contain nodes from different models, creating a mixed model plan.



(a) After model selection, elevate remaining nodes (node after failure to original goal) to match model selected. Plan to goal set (1,2,3,4, and original goal)



(b) Successful re-plan to intermediate goal 3, remainder of plan sent from there to original goal

Figure 2: Localizing around the infeasible area by planning to intermediate goals, in the new selected model, rather than re-planning to the original goal.

Algorithm 3 To plan to intermediate goals it is necessary to elevate the planning model of the remaining nodes in the plan, after the infeasibility, to be the same as the current model. This allows the planner to plan to a possible set of goals rather than a single goal.

```
1: procedure SETGOALS
2:     node = findNodeB4Repair(globalPlan);
3:     m = node.getModel();
4:     setGoals(translateRemainingNodes(globalPlan,m));
5: end procedure
```

4 Implementation

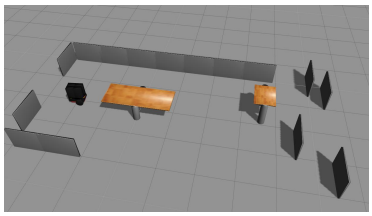
This section presents the models from the motion planning community that we used for our experiments. We also show translation between the models and how collision checking varies based on the model. Section 4.2 describes the low-level robot controller, and the collision monitoring techniques we applied for execution.

In our current implementation, the real world is represented by the Gazebo simulator (http://gazebosim.org/) which models dynamics by simulating rigid-body physics. Our test environment contains three dimensional overhangs of different heights as well as a sliding door that periodically opens and closes every 10 seconds. Pictures of the door open and closed are shown in Figure 4 (a) and Figure 4 (b). We are setting up an environment more complex than the lower models can handle for the sake of demonstrating the benefits of different model use.

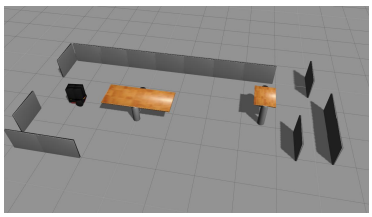
Plans are generated using the Open Motion Planning Library (OMPL) (Şucan, Moll, and Kavraki 2012). While

Algorithm 4 A global plan saves the proper model with each plan node. Partial paths are merged back into the global plan. When planning to intermediate goals the remainder of the old plan must also be added to the partial plan.

```
1: function SAVEPLAN(p, globalPlan)
2:     if globalPlan!=empty then
3:         planPrepend = findPartial(p.start(), p.end(), globalPlan)
4:         planRemainder = findPartial(p.end(), globalPlan);
5:         globalPlan = planPrepend + p + planRemainder;
6:         addConnectionPoint(planPrepend.end(), p.start());
7:         addConnectionPoint(p.end(), planRemainder.start());
8:     else
9:         globalPlan = p;
10:    end if
11:    return globalPlan
12: end function
```



(a) Open door



(b) Closed door

Figure 4: The world with a door that opens periodically on the right side (every 10 seconds). A large center overhang the robot cannot go under, and a smaller overhang the robot can go under.

the underlying motion planner is not so important, we use Rapidly Exploring RandomTrees (RRTs) to generate motion plans (LaValle and Kuffner 2001) since RRTs can easily handle various models, including those with differential constraints and dynamics. We defined the RRT distance function to primarily use the x and y components.

For intermediate goals, θ is added to the distance function to make sure mixed plans with connection points for θ-models align properly with previous plans. Also, as a heuristic in RRT planners, the goal is sampled with some probability. We probabilistically weight the remaining intermediate goals linearly based on the inverse of their distance from the repair segment start. This biases intermediate goals to those closest to the detected impediment in order to re-use more of the remaining plan when possible.

Figure 3 shows the plan-time algorithm generation for the navigation motion planning domain. Each generation stage of the plan is shown resulting in a final plan which merges

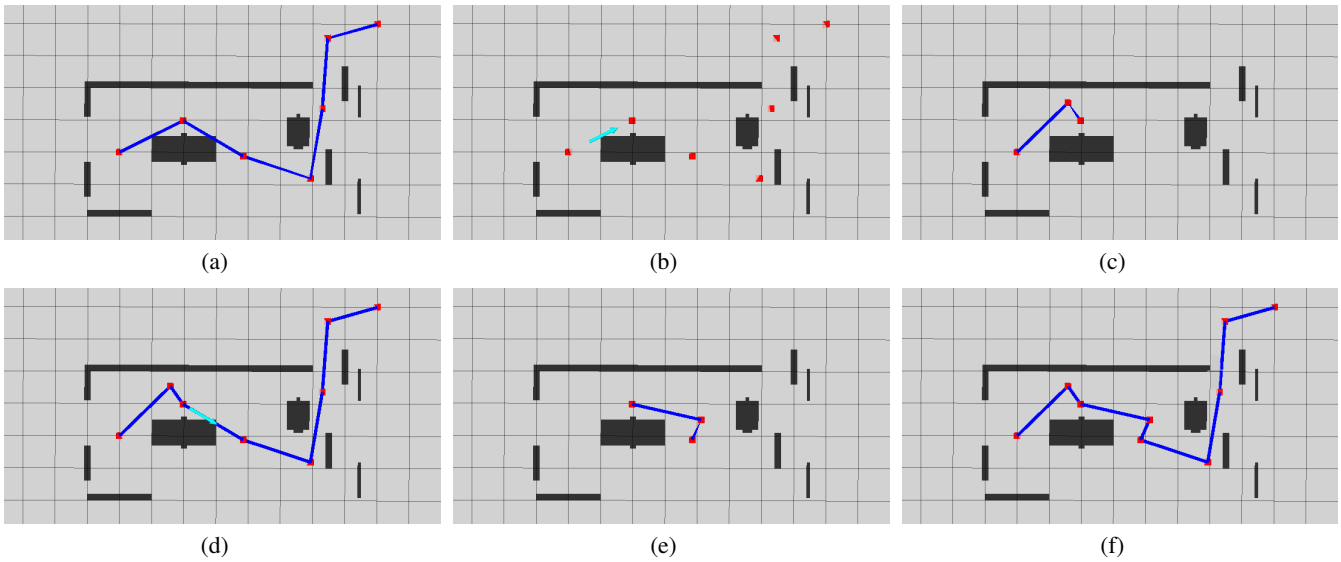


Figure 3: The dark blue lines connect the waypoints in the path. A plan is generated in the lowest fidelity space (a). The plan is translated into the highest model and a collision is detected in the plan run in the highest model (b). The light blue arrow shows the detected collision. The model selector re-plans in a higher selected model (this case [x,y,z]) (c). This partial plan is merged back into the global plan (d), where another collision is detected when checked in the highest model. The model selector re-plans a partial plan in the selected model[x,y,z] (e). The final mixed fidelity plan with no more detected repairs is constructed (f). Note: overhangs projected for illustration purposes.

partial plans from different models.

4.1 Models Used

Our robot has only one subsystem, the chassis, which is differential drive and can translate (in 2D x, y space where x and y are coupled) and rotate in theta (x, y, θ space). The model state focuses on positions in x, y, z space and SO2 space. Time is added to the state space for models that have velocities, which allows us to represent dynamic obstacles that can change position periodically.

For the action space, the underlying ordinary differential equations specific to a non-holonomic two wheeled vehicle were constructed based on the unicycle model (LaValle 2006). Models with proper velocity space sample the right and left wheel velocities separately rather than just sample a linear velocity. Table 1 lists the state, controls, and action spaces for the models we implemented. The models we selected are chosen explicitly to illustrate the benefits of switching. Models for real world robot use are expected to be more sophisticated. Figure 5 shows different plans generated using different models.

The robot's model fidelity changes in two ways. The fidelity of the physical robot changes for collision checking during planning, and the fidelity of robot motions vary to better capture the underlying controller. In particular, models without z treat the environment as 2D, in terms of collision checking, while models with z do full 3D collision checking. [3]

[3]Note: For our experiments we cut the z-dimension at a particular low height. The overhangs are not seen in the planner for the [x,y] model in Figure 3 during step (a). An alternative is to project the z-dimension into 2D. Projection is a more conservative approach since the robot would never consider going under the overhangs in [x,y]. In both cases, there is information lost to models that do not consider the z-dimension.

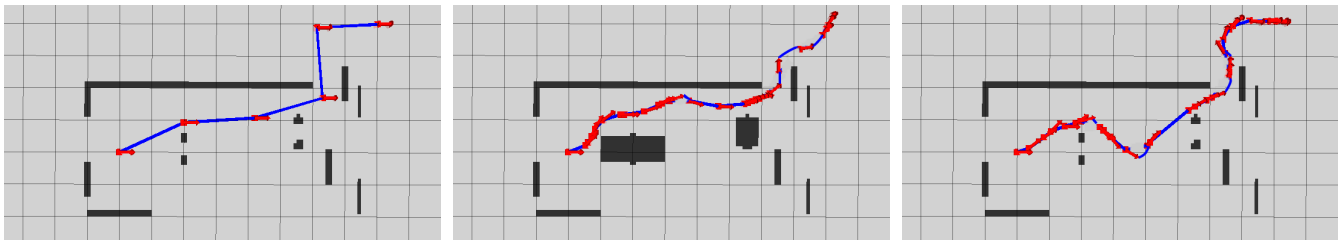
Collision Checker We check collisions only if obstacles inflated by the robot's bounding sphere intersect the current state. Table 2 describes how collision checking changes for different models.

For models with velocity, we use an additional time variable in the state to properly index the environment we are checking against (doors that are closed or open). Models without velocity assume the doors are open.

To help account for uncertainty in the robot's motions, we create a small buffer around the robot by increasing the robot footprint 6% in the x and y directions (3% on each side).

Translating Between Models Translation functions exist to convert states and actions in lower fidelity models to higher fidelity (see Table 3 for details). For models that generate actions that are not directly executable by the controller, additional controls are added. This is necessary for the purely geometric models that have planar motions. For instance, the [x,y] model space generates plans that have the robot turn-in-place and follow a straight line to the next waypoint. The translation function adds extra waypoints to enable the controller to perform turn-in-place actions.

For checking translated plans in higher models we use the same propagation function used in the RRT to plan between waypoints. This forces the use of the same motion equations and collision checker as the model that is being checked in.



(a) x, y (2d environment, planar motions) (b) x, y, z, θ (3d environment, constrained s-curve motions) (c) $x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}$ (2d environment + time, smoother unconstrained s-curve motions)

Figure 5: Examples of plan generation for three different models.

Table 1: Models Used

Model label	States	Control Inputs	Action Space
[x, y]	x and y	none	straight line motions and interpolation.
[x, y, z]	x, y, and z	none	straight line motions and interpolation.
[x, y, θ]	x, y, and theta	sample linear velocity, u[0], rotational velocity u[1] found by dividing by radius.	Equations of motion. $\dot{x} = u[0]\cos(\theta)$ $\dot{y} = u[0]\sin(\theta)$ $\dot{\theta} = u[1]$
[x, y, z, θ]	x, y, z, and theta	sample linear velocity, u[0], rotational velocity u[1] found by dividing by radius.	Equations of motion. $\dot{x} = u[0]\cos(\theta)$ $\dot{y} = u[0]\sin(\theta)$ $\dot{\theta} = u[1]$
[x, y, θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$]	x, y, theta, and time	sample left and right wheel velocities u[0], u[1].	Equations of motion. $\dot{x} = \frac{(u[0]+u[1])}{2\cos(\theta)}$ $\dot{y} = \frac{(u[0]+u[1])}{2\sin(\theta)}$ $\dot{\theta} = u[1] - u[0]$
[x, y, z, θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$]	x, y, z, theta, and time	sample left and right wheel velocities u[0], u[1].	Equations of motion. $\dot{x} = \frac{(u[0]+u[1])}{2\cos(\theta)}$ $\dot{y} = \frac{(u[0]+u[1])}{2\sin(\theta)}$ $\dot{\theta} = u[1] - u[0]$

Since the global plan contains a mix of models, the translation function loops through the global plan translating partial plans between connection points.

4.2 Execution

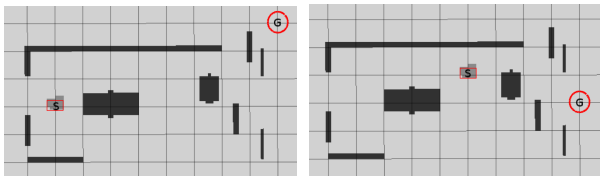
Robot Controller Angular and linear velocity controls are sent for the robot to execute. Controls are matched by having separate PID controllers for each wheel. The robot executes these controls until it is within epsilon of the next waypoint or it crosses a line segment that goes through the next waypoint perpendicular to the robot's heading. The robot follows smoother curves by modifying the commanded linear velocity to be a function of the angular error when the angular error is greater than some epsilon.

The controller also uses a correction in angular velocity for better path following. It adjusts the robot to turn towards the next waypoint when within some small linear distance (0.6) and then turn towards the next planned theta within an even smaller linear distance (0.1).

Table 2: Collision Checking

Model label	States	Collision Checking
[x, y]	x and y	in x and y, with 2D obstacles.
[x, y, z]	x, y, and z	in x, y, and z with 3D obstacles.
[x, y, θ]	x, y, and theta	in x, y, and theta with 2D obstacles.
[x, y, θ]	x, y, z, and theta	in x, y, z, and theta with 3D obstacles.
[x, y, θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$]	x, y, theta, and time	in x, y, and theta with 2D obstacles indexed by time.
[x, y, z,θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$]	x, y, z, theta, and time	in x, y, z, and theta with 3D obstacles and indexed by time.

Execution Monitor We use a Gazebo contact sensor to simulate the robot's bump sensor. If the robot bumps into anything a failure message is sent to indicate failure during execution. This failure signal can be used to initiate replanning, using a variant of Algorithm 1.



(a) Test 1 start and end (b) Test 2 start and end

Figure 6: The start and end positions for the simulation runs.

5 Experimental Results

We ran experiments in the simulator world for two different start and end states as shown for Test1 and Test2 in Figure 6. Test1 requires the robot to traverse through more obstacles, take more turns, and navigate more overhangs than Test2. Test2 focuses on the dynamic sliding door. It places the robot closer to the doors and changes the goal to be directly behind the doors.

Table 3: Translation

Model label	Translate to Label	Translation
[x,y]	[x,y,z]	add z
[x,y]	[x,y,θ]	add theta by finding arctan from next waypoint and get linear and angular velocity from distance to next waypoint.
[x,y,z]	[x,y,z,θ]	add theta by finding arctan from next waypoint and get linear and angular velocity from distance to next waypoint.
[x,y,θ]	[x,y,z,θ]	add z.
[x,y,θ]	[x,y,θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$]	convert linear and angular velocity to left and right wheel velocities. Time variable calculated through state propagation.
[x,y,z,θ]	[x,y,z,θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$]	convert linear and angular velocity to left and right wheel velocities. Time variable calculated through state propagation.
[x,y,θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$]	[x,y,z,θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$]	add z.

We ran three test types. First we ran all six models independently without allowing switching, as shown in Table 4. Then we demonstrate our approach using Algorithm1 and always starting in the lowest model. We refer to these results as 'switch all' shown in Table 5. Lastly, for comparison, we ran tests that only switch between the lowest and highest model, again always starting in the lowest model.

For all tests we recorded the planing time mean and std deviation for 100 runs. We also display the percentage of successful executions that occurred for running the final produced plan in simulation, without execution time replanning.

To produce good plans, we generated 30 RRT plans and chose the best (shortest) one for execution.

Note that we are constructing scenarios that are more likely to fail in the lower fidelity models to show the efficacy of our approach. In normal practice we would expect the initial model (in this case [x,y]) to fail rarely.

Table 4 shows the percentage of successful executions to the goal for each model (without switching) as well as their average planning times. We see that as the models increase in fidelity the mean plan-time over 30 runs increases. The time change in adding the z-component is less than that of theta since this is purely a geometric change which has collision checking in three-dimensions rather than two. The search space does not change. For models that incorporate theta, collision checking is done using theta. Additionally, theta is sampled and the linear velocity is sampled to constrain motion to s-curves. This increases the state search space for waypoint targets and the action space. The two highest fidelity models take the longest since they sample both the right and left wheel velocities as well as incorporating time in the state which adds additional collision checks for sliding doors. This also creates a larger search space which increases the branching factor causing higher planning times.

Test1 starts in front of a long overhang. Models that include the z-dimension are the most beneficial for this test. Additionally, since this test ends near the sliding door velocity models played a minor role in task success.

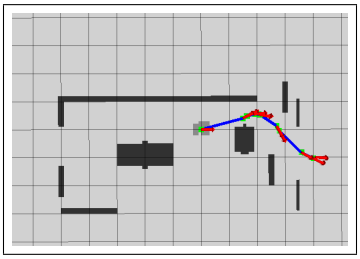


Figure 7: An example of a plan generated in model $[x, y, z, \theta, \dot{x}, \dot{y}, \dot{\theta}]$ for Test2.

In Test2, success rates are much higher for models with velocity since they do properly model the time space (note that models that do not consider time assume the sliding doors are always open). Figure 7, is an example of a path generated in the model [x,y,z,$\theta, \dot{x}, \dot{y}, \dot{\theta}$]. Note that the velocity decreases as the robot nears the doors (as evidenced by the smaller distance between waypoints), waiting for the doors to open, and then increases again to rush through the doors before they close.

Table 4: Results for single model runs with no switching.

Model	success %	plan-time mean(s)	plan-time std dev
Test 1 (From x=-2, y=-2 to x=6.0, y=2.0). Includes overhangs.			
[x,y]	19%	0.13	0.01
[x,y,θ]	20%	79.0	25.5
[x,y,z]	67%	1.79	0.52
[x,y,z,θ]	87%	93.0	19.1
[x,y,θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$]	17%	110.2	30.2
[x,y,z,θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$]	90%	158.5	43.3
Test 2 (From x=2.0 , y=-1.0 to x=6.0 to y=-2.0). Includes moving door.			
[x,y]	35%	0.10	0.01
[x,y,θ]	46%	56.1	14.6
[x,y,z]	44%	1.22	0.33
[x,y,z,θ]	52%	81.1	19.5
[x,y,θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$]	73%	162.2	51.1
[x,y,z,θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$]	75%	194.2	58.2

Table 5: Switching results.

Model	success %	plan-time mean(s)	plan-time std dev
Test 1 (From x=-2, y=-2 to x=6.0, y=2.0). Includes overhangs.			
Switch all	90%	2.44	8.44
Switch lowest and highest	86%	17.4	23.0
Test 2 (From x=2.0 , y=-1.0 to x=6.0 to y=-2.0). Includes moving door.			
Switch all	85%	128.2	127.7
Switch lowest and highest	80%	149.4	146.3

Table 5 shows the results of our switching algorithm. Note that in these results mean planning time also includes the time it took for selecting the next appropriate model to replan in. However, model selection times are negligible. Our

algorithm, 'switch all', produces success rates comparable to that of the highest fidelity model, but with much more efficient planning times. For comparison, we also evaluated a more traditional approach where the system switches between only two models (the lowest and highest fidelity models). The traditional approach produces success rates comparable to our full switching approach, but the plan times are higher.

The plan-time gains for switching were higher for Test1 than Test2. This is due to which models were selected during plan-time to switch to. The models selected for switching with Test2 were most often the velocity models since infeasibility was often detected in the time dimension. This causes the times to be higher on average. Additionally, plans with impediments detected directly in front of the sliding doors tend to take much longer to repair since the robot cannot travel backwards. Test1 switched more often to lower models with the 'z' dimension which gave a larger plan-time savings. This is most evident when compared with the switching between only the highest and lowest model. In many parts of the Test1 space the highest model is not necessary for task success.

In general, switching creates plans with mean times lower than the highest fidelity model and higher than the lowest fidelity model. Intermediate goals further improve planning times. This is evident in the results for switching between the low and high model in Test1. The mean planning time is much lower because the highest model is only necessary for a short detour in the space. Switching with all models created plans with mean times lower than just switching between the low and high models without sacrificing execution success. The switching tests also contained a handful of cases which failed to generate a plan. This suggests additional evaluations for completeness and guarantees.

The fact that the robot still fails in our highest model during execution 20-25% of the time means the framework can still be extended to additional models. For instance, assuming instantaneous velocity and acceleration caused the robot controller to vary from the plan. A model that can capture this would reduce the failure rates even more. In cases where it is difficult to model the information necessary to approximate the underlying controller, it would also be possible to add uncertainty.

6 Future Work

The architecture allows for expansion to additional models. The model graph can include other robot subsystems such as models for a manipulator, or those that combine manipulator and chassis motions. We would also like to include models with simple physics such as friction for object interaction, and forces for pushing. The architecture allows models that are intractable over the full plan generation to still be used locally which would increase robot robustness by enabling even higher success rates.

We observed that if the robot controller did not follow the generated plan exactly time state synchronization drifts. If this occurred when the robot was passing near the sliding doors, during a state change, it most certainly caused a failure. At other times it had little effect. The ability to add uncertainty as a buffer around time, would allow the robot to be more conservative when deciding to pass the doors. Therefore, we would like to investigate how uncertainty fits into the model hierarchy and selection process. We recognize increasing uncertainty, such as a very large robot footprint, can cause the robot to be conservative and think it cannot traverse through a narrow area. This is why we would also like to investigate different levels of uncertainty coverage. We believe varying uncertainty is another form of varying fidelity and will give the robot more choices to increase success when applicable.

Lastly, we would like to combine the execution-time version of the approach with our plan-time approach for real-robot application. Execution time is important for obtaining information that is not available during plan-time. This could be due to uncertainty in the world or sensing abilities. An execution time approach would focus the algorithm towards the failure recovery domain. Information acquired during execution is then provided to the planner allowing the robot to switch to models it previously did not have enough information for. This would continue to provide a more robust plan.

7 Conclusions

We present an approach that leverages a multi-fidelity model graph to produce a mixed-model plan. This plan finds a good balance between decreased planning time and increased robustness by giving the robot the ability to re-plan in a more detailed model to provide a more accurate representation of reality. Just as the robot's operation space is non-uniform, containing a mix of simple and complex areas, our algorithm tries to capture the space with a mix of low and high fidelity models generating an efficient plan that does not sacrifice execution success.

Our tests show that the algorithm improves computation time while obtaining comparable performance to planning always in the highest fidelity model. The average overall planning time (initial plus re-plans) is greater than the average time for [x,y] planning alone, but still less than the average for the higher fidelity spaces. Switching between multiple models is also cheaper than just switching between the lowest and highest model. Our switching tests localize planning around the infeasible location without sacrificing the probability of successful executions.

The results also show there is not always a single best model to use, but rather that it depends on the situation. For example, even though modeling the differential constraints of the robot is higher fidelity than a purely geometric model, that model will not help if the real problem is not considering the z dimension (e.g. if overhangs exist in the world). This is why it is important to have a separate model selection stage that can reason about which model should be used for repair. This is increasingly important as robots, their tasks, and their environments become more complex. We believe that, especially as these complexities increase, selecting the most appropriate model to plan in is important for robust, tractable planning.

References

Barbehenn, M., and Hutchinson, S. 1995. Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest paths trees. *Robotics and Automation, IEEE Transactions on* 11(2):198–214.

Behnke, S. 2004. Local multiresolution path planning. In *Robocup 2003: Robot Soccer World Cup VII*. Springer. 332–343.

Bruce, J., and Veloso, M. 2002. Real-time randomized path planning for robot navigation. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 3, 2383–2388. IEEE.

Choi, W.; Zhu, D.; and Latombe, J.-C. 1989. Contingency-tolerant robot motion planning and control. In *Intelligent Robots and Systems' 89. The Autonomous Mobile Robots and Its Applications. IROS'89. Proceedings., IEEE/RSJ International Workshop on*, 78–86. IEEE.

Dogar, M. R.; Hsiao, K.; Ciocarlie, M.; and Srinivasa, S. S. 2012. Physics-based grasp planning through clutter. In *In RSS*. Citeseer.

Ferguson, D.; Kalra, N.; and Stentz, A. 2006. Replanning with rrts. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, 1243–1248. IEEE.

Fernández-Madrigal, J.-A., and González, J. 2002. Multihierarchical graph search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 24(1):103–113.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial intelligence* 3:251–288.

Göbelbecker, M.; Gretton, C.; and Dearden, R. 2011. A switching planner for combined task and observation planning. In *AAAI*.

Gochev, K.; Cohen, B.; Butzke, J.; Safonova, A.; and Likhachev, M. 2011. Path planning with adaptive dimensionality. In *Fourth Annual Symposium on Combinatorial Search*.

Gochev, K.; Safonova, A.; and Likhachev, M. 2012. Planning with adaptive dimensionality for mobile manipulation. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, 2944–2951. IEEE.

Gochev, K.; Safonova, A.; and Likhachev, M. 2013. Incremental planning with adaptive dimensionality. In *Twenty-Third International Conference on Automated Planning and Scheduling*.

Hauser, K., and Latombe, J.-C. 2009. Multi-modal motion planning in non-expansive spaces. *The International Journal of Robotics Research*.

Hauser, K.; Ng-Thow-Hing, V.; and Gonzalez-Baños, H. 2011. Multi-modal motion planning for a humanoid robot manipulation task. In *Robotics Research*. Springer. 307–317.

Howard, T. M., et al. 2009. Adaptive model-predictive motion planning for navigation in complex environments.

Kaelbling, L. P., and Lozano-Pérez, T. 2011. Hierarchical task and motion planning in the now. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 1470–1477. IEEE.

Kambhampati, S., and Davis, L. 1986. Multiresolution path planning for mobile robots. *Robotics and Automation, IEEE Journal of* 2(3):135–145.

Knepper, R. A., and Mason, M. T. 2011. Improved hierarchical planner performance using local path equivalence. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, 3856–3861. IEEE.

Koenig, S., and Likhachev, M. 2002. D* lite. In *AAAI/IAAI*, 476–483.

LaValle, S. M., and Kuffner, J. J. 2001. Randomized kinodynamic planning. *The International Journal of Robotics Research* 20(5):378–400.

LaValle, S. M. 2006. *Planning algorithms*. Cambridge university press.

Pivtoraiko, M., and Kelly, A. 2005. Efficient constrained path planning via search in state lattices. In *International Symposium on Artificial Intelligence, Robotics, and Automation in Space*.

Pivtoraiko, M., and Kelly, A. 2008. Differentially constrained motion replanning using state lattices with graduated fidelity. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, 2611–2616. IEEE.

Plaku, E.; Kavraki, E.; and Vardi, M. Y. 2010. Motion planning with dynamics by a synergistic combination of layers of planning. *Robotics, IEEE Transactions on* 26(3):469–482.

Raibert, M.; Blankespoor, K.; Nelson, G.; Playter, R.; et al. 2008. Bigdog, the rough-terrain quadruped robot. In *Proceedings of the 17th World Congress*, volume 17, 10822–10825.

Seegmiller, N., and Kelly, A. 2014. Enhanced 3d kinematic modeling of wheeled mobile robots.

Steffens, R.; Nieuwenhuisen, M.; and Behnke, S. 2010. Multiresolution path planning in dynamic environments for the standard platform league. In *Proceedings of 5th Workshop on Humanoid Soccer Robots at Humanoids*.

Stentz, A. 1995. The focussed dˆ* algorithm for real-time replanning. In *IJCAI*, volume 95, 1652–1659.

Stephens, B. 2007. Humanoid push recovery. In *Humanoid Robots, 2007 7th IEEE-RAS International Conference on*, 589–595. IEEE.

Sucan, I. A., and Kavraki, L. E. 2011. Mobile manipulation: Encoding motion planning options using task motion multigraphs. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 5492–5498. IEEE.

Şucan, I. A.; Moll, M.; and Kavraki, L. E. 2012. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine* 19(4):72–82. http://ompl.kavrakilab.org.

Wolfe, J.; Marthi, B.; and Russell, S. J. 2010. Combined task and motion planning for mobile manipulation. In *ICAPS*, 254–258.

Mixed Discrete-Continuous Heuristic Generative Planning based on Flow Tubes (extended version)

Enrique Fernandez-Gonzalez, Erez Karpas and Brian C. Williams
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
32 Vassar Street, Building 32-224, Cambridge, MA 02139
efernan@mit.edu, karpase@mit.edu, williams@mit.edu

Abstract

Nowadays, robots are programmed with a mix of discrete and continuous low level behaviors by experts in a very time consuming and expensive process. Existing automated planning approaches are based on hybrid model predictive control techniques, which don't scale well due to time discretization, or temporal planners, which sacrifice plan expressivity by only supporting discretized fixed rates of change in continuous effects. We introduce Scotty, a mixed discrete-continuous generative planner that finds the middle ground between these two. Scotty can reason with linear time evolving effects whose behaviors can be modified by bounded control variables, with no discretization involved. Our planner exploits the expressivity of continuous effects represented by flow tubes, and the performance of heuristic forward search. The generated solution plans are better suited for robust execution, as executives can use both time and continuous control variables to react to disturbances.

Introduction

Robotic missions are commonly programmed by highly skilled experts in an expensive and time consuming process. As robotic systems become increasingly more common, it is desirable to devise more efficient systems to program the behavior of these robots. This often involves reasoning over a mix of discrete and continuous conditions and effects with temporal deadlines and constraints.

When no discrete conditions or effects need to be considered, robotics planning reduces to trajectory optimization, that is, coming up with a control sequence over time that satisfies a set of feasibility constraints and maximizes an objective. Model-predictive control is, perhaps, the most common approach, which frames trajectory planning as a discrete time continuous state optimization problem. Trajectory optimization has been generalized to a hybrid problem in which control behaviors are engaged and disengaged, and control trajectories are generated for those different behaviors. Kongming (Li and Williams 2008; Li 2010; Li and Williams 2011) provides one such approach that generalized the discrete time, continuous model-predictive control framework to a mixed discrete-continuous formulation using an encoding based on flow tubes and hybrid flow graphs. Although quite expressive, Kongming is limited to small horizons due to the size of the corresponding optimization problem.

On the other hand, the AI planning community has developed an extensive set of discrete temporal planners whose performance has improved substantially over the last decade. Such planners have been used successfully in real robotic missions by NASA (Muscettola et al. 1998) and others, but continuous effects have often been neglected in the planning stage and only considered during plan execution. The planning community has also extended traditional planners to deal with continuous variables. Heuristic forward search variants of these planners such as COLIN (Coles et al. 2009) have demonstrated good empirical performance on IPC benchmarks. Some of these planners have been applied to robotics problems. For example, POPF (Coles et al. 2010) has been used to design AUV inspection plans of underwater installations (Cashmore et al. 2014). However, the planner did not explicitly consider the continuous motion of the robot, but instead chose a mission path by selecting discrete waypoints that were previously generated using random sampling motion planning techniques. The benefit of this type of planners is that time is not discretized, enabling the planner to handle long time horizons. The challenge, however, is that although continuous linear time evolving effects are supported, actions have constant control values. While trajectory optimization can be mimicked by creating multiple copies of each action with discretized control values, this approach does not scale well.

Kongming generates control trajectories that preserve the richness of classical model predictive control techniques, but at the severe cost of computational efficiency. On the other hand COLIN leverages the efficiency of heuristic forward search methods at the cost of the richness of the control trajectories generated. Finding the middle ground that preserves essential elements of the expressivity of Kongming, while preserving the efficiency offered by COLIN is desirable for robotic applications. In this paper, we present Scotty, a mixed discrete-continuous temporal planner temporal planner that combines the representation of continuous effects based on flow tubes from Kongming, with the efficient solving method based on heuristic forward search and linear programs for consistency checking that COLIN uses. Scotty leverages the temporal flexibility of temporal planners and the control trajectory flexibility from

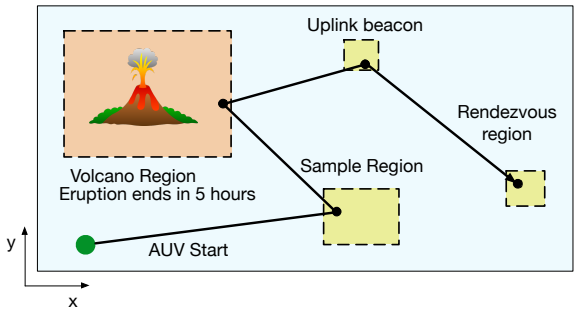


Figure 1: Example AUV mission.

t	Event	State
0.0000	START-VOLCANO-ERUPTION	x=0.0, y=0.0
0.0001	START-DO-MISSION	x=0.0, y=0.0
0.0002	START-NAVIGATE-AUV	x=0.0, y=0.0
100.0002	END-NAVIGATE-AUV	x=400.0, y=75.0
100.0003	START-TAKE-SAMPLE	x=400.0, y=75.0
105.0003	END-TAKE-SAMPLE	x=400.0, y=75.0
105.0004	START-NAVIGATE-AUV	x=400.0, y=75.0
211.2504	END-NAVIGATE-AUV	x=150.0, y=500.0
289.9999	START-COLLECT-VOLCANO-DATA	x=150.0, y=500.0
299.9999	END-COLLECT-VOLCANO-DATA	x=150.0, y=500.0
300.0000	END-VOLCANO-ERUPTION	x=150.0, y=500.0
300.0001	START-NAVIGATE-AUV	x=150.0, y=500.0
337.5001	END-NAVIGATE-AUV	x=300.0, y=545.0
337.5002	START-UPLINK-VOLCANO-DATA	x=300.0, y=545.0
342.5002	END-UPLINK-VOLCANO-DATA	x=300.0, y=545.0
342.5003	START-NAVIGATE-AUV	x=300.0, y=545.0
430.0003	END-NAVIGATE-AUV	x=650.0, y=420.0
430.0004	START-ASCEND	x=650.0, y=420.0
440.0004	END-ASCEND	x=650.0, y=420.0
440.0005	END-DO-MISSION	x=650.0, y=420.0

Table 1: Example solution for the motivating scenario.

model predictive control techniques to generate plans that are suited for robust execution.

Motivating Example

To motivate the need for a mixed discrete-continuous planner for robotic missions, we introduce an example scenario involving a scientific autonomous underwater vehicle (AUV) mission (Figure 1). In this mission the AUV has a time frame to complete two objectives before ascending to the rendezvous point. The AUV needs to take a sample in an specified interest region and collect data of an ongoing underwater eruption site before the eruption ends, in five hours. The collected data needs to be uplinked to an optical underwater beacon, so that nearby scientists can analyze it as soon as possible.

Note that this mission requires activities with discrete effects, such as *collect-data*, but also activities with continuous effects, such as *navigate*. In effect, in this mission we define the continuous state variables x and y that specify the position of the AUV at all times. Activities such as *collect-data* require the AUV to be in a specified region (constraints on x and y). The AUV can change its position using the continuous *navigate* activity. This activity varies the x and y state variables with time according to the **control variable** *speed*, which is bounded and can be modified continuously. The hybrid planner needs to be able to reason with these continuous effects in order to take the AUV to the regions in which samples or data can be collected, and do it within their respective time windows. Table 1 shows an example fixed solution plan for this mission. Figure 2 shows the state trajectory and values for the control variables from this solution. In a later section, we discuss how to generate a flexible plan suitable for robust execution from the fixed solution plan for this problem.

In the next sections we describe Scotty's problem statement, its solution and how that solution is found.

Problem Statement

The input to the problem consists of the *domain*, *initial conditions* and *goal*. These extend PDDL 2.1 (Fox and Long 2011) with some modifications that allow us to define activities with continuous effects that depend on *bounded control variables*. Similarly to PDDL 2.1, durative activities have a bounded controllable duration, discrete effects and continuous and discrete conditions defined *at start*, *over all* and *at end*. Continuous conditions are defined by linear inequalities over the state variables according to:

$$\mathbf{c^T x'} \leq 0 \tag{1}$$

, where $\mathbf{x'} = (x_1, \ldots, x_{n_x}, 1)^T$ and $\mathbf{c} \in \mathbb{R}^{n_x+1}$ is a vector of coefficients, with n_x being the number of state variables of the system.

Our problem statement differs from PDDL 2.1 in the effects on continuous variables. Each activity has a set of control variables, which can be seen as continuous parameters — each of those constrained by lower and upper bounds. The continuous effects of the activity are similar to the continuous effects of PDDL 2.1, except they are affected by the value chosen for the control variables. We restrict each continuous effect to involve only a *single* control variable, c_{var}, and thus each continuous effect can be defined by $\langle x, c_{var}, k \rangle$, where x is a state variable, c_{var} is a control variable, and k is a constant.

In the simple case, where a single continuous effect $\langle x, c_{var}, k \rangle$ is active from time t_{start} to time t_{end} with c_{var} fixed to a constant value of c throughout the duration, then $x(t)$, the value of state variable x at time t is defined by $x(t) = x(t_{start}) + k \cdot c \cdot (t - t_{start})$ with $t_{start} \leq t \leq t_{end}$.

Multiple continuous effects on the same state variable are additive, and thus $x(t)$ is defined by:

$$x(t) = x(0) + \int_0^t C_x(\tau) d\tau \tag{2}$$

where $C_x(t)$ is the sum of the values of control variables in active continuous effects modifying x at time t (represented by the set E).

$$C_x(t) = \sum_{\langle x, c_{var}, k \rangle \in E} k \cdot c_{var}(t) \tag{3}$$

where $c_{var}(t)$ denotes the value chosen for the control variable c_{var} at time t. The *navigate* activity of the example AUV mission is shown in Figure 3. Note the bounded *control variables* velX and velY.

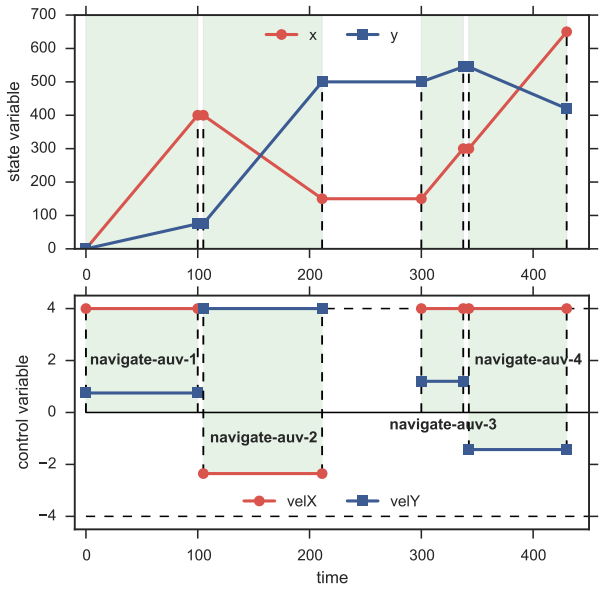


Figure 2: Trajectories of x and y in the example problem due to the four *navigate* activities and the values of their control variables *velX* and *velY* extracted from the fixed plan solution from Table 1.

```
(:durative-action navigate
:control-variables ((velX) (velY))
:duration (and (<= ?duration 5000))
:condition (and
                (over all (>= (velX) -4)) (over all (<= (velX) 4))
                (over all (>= (velY) -4)) (over all (<= (velY) 4))
                (over all (<= (x) 700)) (over all (>= (x) 0))
                (over all (<= (y) 700)) (over all (>= (y) 0))
                (at start (AUV-ready)))
:effect (and
           (at start (not (AUV-ready)))
           (at end (AUV-ready))
           (increase (x) (* 1.0 (velX) #t))
           (increase (y) (* 1.0 (velY) #t))))
```

Figure 3: Navigate activity modified by continuous control variables *velX* and *velY*.

The problem initial conditions are given by the set of *true* propositions (P_0) and an assignment to the state variables at the start ($x_i(0)$). Finally, the *goal* consists of the discrete and continuous conditions that need to hold at the end.

A fixed solution plan to the planning problem consists of a list of scheduled activities defined by the following properties: activity a, execution start time t_a, duration d_a, and a trajectory of the value of each control variable of a from t_a to $t_a + d_a$, that is a function giving the value of each control variable at any given moment, $c_{aj}(t)$ with $t_a \leq t \leq t_a + d_a$. Note that a fixed plan gives us the values of all the state variables (discrete and continuous) at every time t between 0 and the end of the plan — the state trajectory. We further require that all activity conditions are satisfied by the state trajectory at the appropriate times, as in PDDL 2.1.

A fixed plan is not robust: any small deviation during execution will cause the plan to fail. However, it is possible to use flexible plans that allow an executive to respond to small disturbances during execution. One such representation is Qualitative State Plans (QSP) (Léauté and Williams 2005). A QSP is defined by a set of events (starts and ends of activities), along with simple temporal constraints and state constraints. Simple temporal constraints (Dechter, Meiri, and Pearl 1991) are of the form $lb \leq t_j - t_i \leq ub$, where t_i and t_j are execution times of events and lb, ub are a fixed lower and upper bound on the duration between t_i and t_j. State constraints specify legal values for continuous state variables at the start and end events of the state constraint, as well as the legal trajectories of continuous state variables and control variables. These trajectories specify the legal values of these at any moment in time between the events. Typically in robotic applications, these are given by the dynamics of the robot. For Scotty, these are limited to the linear form $\Delta x = c_{var} \cdot \Delta t$ with $c_l \leq c_{var} \leq c_u$.

Executives exist that can control a robot given a QSP as an input (Hofmann and Williams 2015). These executives choose the control variables and the execution times of events online, while ensuring that all constraints are met. Therefore, we focus on generating a flexible plan and leave its dynamic execution to the executive. In the next section, we review key ideas from existing planners that Scotty borrows.

Previous Work

In this section we briefly describe the approaches that Kongming and COLIN use to solve their planning problems. Scotty combines the best of these approaches to solve the hybrid planning problem with continuous control variables.

Kongming

Kongming solves the same planning problem as Scotty: actions are hybrid and their continous effects are modified by bounded continuous control variables. One of the main innovations introduced by Kongming consists in representing continuous effects with **flow tubes**, that are abstractions of the infinite number of trajectories that a continuous action can produce.

Another key innovation introduced by Kongming consists in the introduction of the **Hybrid Flow Graph**, the continuous analog to Graphplan's Planning Graph (Blum and Furst 1997). Hybrid actions connect initial state regions to goal regions after some fixed duration using the flow tubes generated from the system dynamics. Kongming expands the Hybrid Planning Graph with alternating action and fact layers until the goal conditions are non-mutex in the last fact layer. Kongming then encodes the problem as a mixed logic linear (non-linear) program that contains the system dynamics constraints on the state variables and logic constraints on binary variables for the discrete conditions and effects (similar to Blackbox's SAT encoding (Kautz and Selman 1999)). Kongming alternates between trying to solve the ML(N)LP and adding additional layers to the graph until the ML(N)LP solver returns a solution.

Although Kongming's approach is innovative, it suffers from performance degradation issues in medium to large problems due to time discretization, as graph layers are discretized using a fixed time step. In problems in which the

time horizon is moderately large, this involves creating many layers. As the number of layers increases, identifying mutex relations becomes exponentially more complicated. This also slows down significantly the ML(N)LP solver, as each additional layer adds many more additional variables and constraints. In a later section of this document, we illustrate with an example how this can become a problem very fast.

COLIN

COLIN solves temporal planning problems with continuous effects as defined in PDDL 2.1. Hybrid actions can have continuous linear time-varying effects, whose rate of change is specified by a fixed gradient.

In order to solve the planning problem, COLIN uses heuristic forward search with the Enforced Hill Climbing algorithm (EHC). Every search state is tested for consistency with a Linear Program in which the real valued variables are the continuous state variables and the times at which actions are executed. The continuous linear time-varying effects and the temporal relations between actions are encoded as constraints. The linear program is also used at each step of the search to determine the minimum and maximum possible bounds for each state variable in order to prune actions that can't possibly be feasible at that point of the search.

COLIN's heuristic is based on the Temporal Relaxed Planning Graph (TRPG) and delete relaxations. COLIN defines FF's analogous delete relaxation for continuous linear time-varying effects by only allowing state variable bounds to grow as a result of those continuous effects (bounds never get smaller). The heuristic value is the number of actions in the relaxed plan.

Although COLIN is a very efficient and proven planner, it is not expressive enough to solve the kind of robotic problems we want to target. COLIN's continuous actions can only use fixed rates of change, according to the following equation:

$$x(t_{end}) = x(t_{start}) + \text{rate-of-change} \cdot (t_{end} - t_{start}) \quad (4)$$

COLIN's formulation can't handle bounded continuous rates of change (control variables). We can simulate this behavior by creating many equivalent continuous actions with discretized fixed rates of change. However, this solution is problematic as described at the end of this document.

From COLIN we borrow the efficient solving approach based on linear programs for testing plan consistency and heuristic forward search. We also use a modified version of COLIN's heuristic. The key innovation of our approach consists in combining Kongming's more flexible representation of continuous actions based on flow tubes, with COLIN's more efficient solving approach based on heuristic forward search and no time discretization.

Approach

In order to find flexible solution plans as described in the Problem Statement section, Scotty first finds a fixed plan. In this section we describe how Scotty uses flow tubes to represent continuous effects modified by continuous control variables, how fixed plans are found and, finally, how a flexible plan is extracted from the fixed plan.

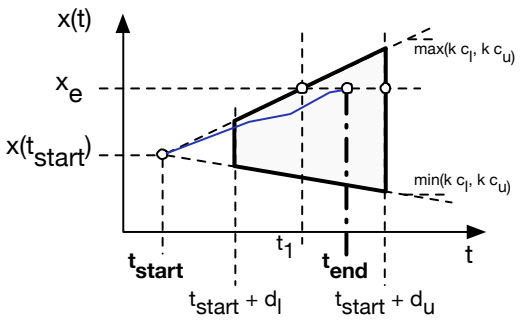


Figure 4: Flow tube with its reachable region (shaded area). The solid blue line represents an example valid state trajectory. The flow tube contains all valid state trajectories.

Flow tube representation of continuous effects

As Kongming, our planner uses flow tubes to represent continuous effects modified by continuous control variables. Each activity can have multiple continuous effects, and each is represented by a flow tube. For Scotty, we limit flow tubes to operate on only one state variable. Flow tubes represent the reachable state space region, that is, the values that the state variable can take after the activity is started. We restrict continuous effects to linear time varying effects.

A flow tube $f(d_l, d_u, c_{var}, x, k)$ is defined by the following properties:

- minimum and maximum duration (d_l, d_u), which are the minimum and maximum durations of the activity the flow tube belongs to.
- state variable x that the flow tube modifies.
- control variable c_{var}, is the activity control variable the flow tube is associated with. Remember from the Problem Statement that control variables are bounded ($c_l \leq c_{var} \leq c_u$).
- the scalar constant factor k that regulates the impact of the control variable on the state variable

If no other flow tubes are affecting state variable x between t_{start} and t_{end}, then the reachable region of x represented by the flow tube $f(d_l, d_u, c_{var}, x, k)$ of an activity executed between t_{start} and t_{end} is defined by the following equations:

$$x(t_{end}) = x(t_{start}) + k \cdot \int_{t_{start}}^{t_{end}} c_{var}(t) \cdot dt \quad (5)$$

with
$$c_l \leq c_{var}(t) \leq c_u \quad (6)$$
$$d_l \leq t_{end} - t_{start} \leq d_u \quad (7)$$

where $x(t_{start})$ is the initial value of the state variable before the activity is executed. Note that, if the value of the

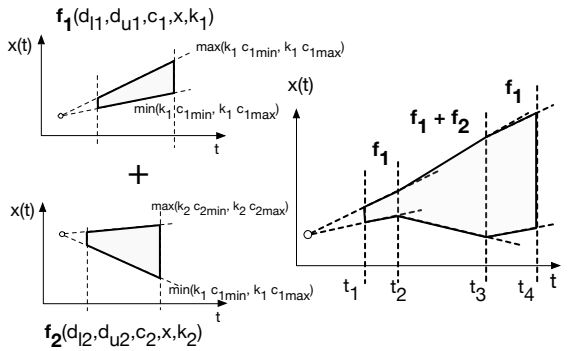


Figure 5: Combined flow tubes.

control variable is constant during the execution of the activity, equation 5 reduces to $x(t_{end}) = x(t_{start}) + k \cdot c_{var} \cdot (t_{end} - t_{start})$.

Figure 4 shows a flow tube. Note that any point in the shaded region (reachable region) can be reached at the end of the activity by carefully choosing the appropriate activity duration and control variable value. In the figure, we can see how the state value x_e can be reached as fast as in $t_{end} = t_1$ if the control variable c_{var} is constant and takes its maximum possible value (u_c), or as late as $t_{end} = t_{start} + d_u$ if $c_{var}(t)$ takes smaller values.

Note that two or more flow tubes operating on the same state variable and belonging to the same or different ongoing activities can be active at the same time, having their effects combined. For example, suppose that activity a_1 has start and end times t_1 and t_4 and activity a_2, t_2 and t_3 and that $t_1 < t_2 < t_3 < t_4$ so that a_2 happens while a_1 is in progress. Suppose also that among their continuous effects, they each have flow tubes f_1 and f_2 respectively that operate on state variable x. Figure 5 shows how the effects of these flow tubes are combined. From the start, any point in the shaded region can be reached. Note that for $t \in [t_2, t_3]$ (while both $a1$ and $a2$ are being executed), the effects of f_2 are added to those of f_1.

In general, if F represents the set of flow tubes belonging to ongoing activities in $t \in [t_a, t_b]$, the following state evolution constraint between t_a and t_b holds:

$$x(t_b) = x(t_a) + \int_{t_a}^{t_b} C_x(\tau) d\tau \quad (8)$$

$$C_x(t) = \sum_{f \in F} k_f \cdot c_{var\,f}(t) \quad (9)$$

where F is the set of flow tubes belonging to ongoing activities in $t \in [t_a, t_b]$.

An important characteristic of flow tubes, as described by Kongming, is that they provide a compact encoding of all feasible trajectories. This property is exploited to find a fixed plan, as explained in the next section.

Finding a fixed solution plan

Now that we've defined how flow tubes represent continuous effects, we proceed to describe how fixed plans are found. In order to do that, Scotty uses a method based on heuristic forward search and linear programs for consistency checking, that is borrowed from COLIN. The main difference is that Scotty's continuous effects are represented by flow tubes modified by control variables, and therefore, the state evolution constraints are different, as will be explained later in this section.

Activities are divided into start and end events, analogous to the start and end snap actions used by many temporal planners (Long and Fox 2003; Coles et al. 2008). In order to find a fixed plan, Scotty needs to find the ordered sequence of start and end events that takes the system from the initial conditions to the goal and the execution time of each event. Scotty also needs to find a trajectory for the values of each activity control variable between the start and end events of the activity ($c_{var}(t)$ with $t_{start} \leq t \leq t_{end}$).

Scotty finds trajectories for the control variables of an activity that are piecewise constant. The number of segments of these trajectories are given by $1 + n_{ev}$, where n_{ev} is the number of events that occur between the start and end events of the activity. Remember that, in general, the value of a control variable can vary within its bounds throughout the execution of the activity. However, Scotty only needs to find one solution at this step. It can be proved that, if the planning problem has a solution, there exists a control variable piecewise constant solution, and that's the fixed solution plan that Scotty finds. The reason is that the state variables of the system only change due to the continuous effects of executing activities and are only constrained by the start, end and overall conditions of activities. Therefore, the constraints that state variables are subject to only change at events (starts or ends of activities). Moreover, these constraints are linear (inequalities as in (1)) and, therefore, the set of valid state variable assignments is convex. Because of that, if the problem can be solved, a fixed plan containing piecewise constant trajectories for the control variables is always a solution. Similarly, if no piecewise constant solution can be found, then the problem doesn't have any solutions. This property is convenient, as we'll show that a linear program formulation exists that can find such piecewise constant solutions.

As COLIN, Scotty uses heuristic forward search to find the sequence of start and end events that form a fixed plan, and linear programs to check the consistency of each search state. The search uses Enforced Hill Climbing, which has proven to be effective in this type of problems (Hoffmann and Nebel 2001). However, because EHC is not complete, if no solution is found, Scotty can optionally try again with a complete algorithm such as best-first search.

Search states contain the set of propositions that are known to be true due to discrete effects, and are augmented with the ongoing activities list and the bounds for all state variables. The ongoing activities list keeps track of the activities that have started but not finished at that state and is needed to keep track of the active overall discrete and continuous constraints. The lower and upper bounds for the state variables are used to prune sections of the search tree that

Type	Constraints
Temporal	**Total order of events:** For any pair of consecutive events i and j $$t_j - t_i \geq \varepsilon \quad (10)$$ **Activity duration:** For every activity whose start and end events are i and j $$d_l \leq t_j - t_i \leq d_u \quad (11)$$ where d_l and d_u are the lower and upper bounds of the activity duration.
State	**Activity conditions:** For every start or end event i $$\mathbf{c_k^T x_i'} \leq 0 \ \ \forall \mathbf{c_k} \in C_i \quad (12)$$ where $\mathbf{c_k}$ is a vector of real coefficients, $\mathbf{x_i'} = (x_{1_i}, \ldots, x_{n_{x_i}}, 1)^T$ and C_i is the set of continuous conditions that are active at event i. That is, the start (or end) conditions of the activity, and the overall conditions of activities that started before i but whose end event occurs after i.

Table 2: Temporal and state constraints used in the consistency linear program. t_k represents the execution time for event k and x_{l_k} represents the value of state variable x_l at event k.

are necessarily not feasible. For example, if the start event of a certain activity a requires state variable x to be greater than 7 but the lower and upper bounds of x are 3 and 5 respectively, the search algorithm won't even try to apply this event. However, an event having a $x \geq 4$ condition will be tried in the search. Note that these state variable bounds are calculated for each state variable independently of the others. As a consequence, the fact that a constraint is satisfied by these bounds doesn't mean that the partial plan is necessarily feasible. Whether the partial plan is really feasible is only discovered when the consistency linear program is solved.

Each search state defines a partial plan as a sequence of start and end events so far, and is tested for consistency using a linear program. The partial plan is feasible if the linear program has a solution. In this linear program the decision variables are the event execution times and the values of the state variables at each event. The constraints include activity duration, start, end and overall conditions and state evolution constraints that are built from the current sequence of start and end events. Table 2 shows the temporal and state constraints between events. These constraints are the same ones that COLIN uses and are explained in detail in its journal article (Coles et al. 2012). Scotty needs a different state evolution constraint, though, due to the presence of control variables. This constraint is given by the flow tube reachability equation (5). Because the values of the control variables can change during the activity execution, and the start and end times of the activity are variables of the linear program, this equation is not linear if control variables are decision variables of the linear program. However, we can redefine the reachability region of the flow tube with the following linear inequalities:

$$x_{end} \geq x_{start} + \min(k \cdot c_l, k \cdot c_u) \cdot (t_{end} - t_{start}) \quad (13)$$
$$x_{end} \leq x_{start} + \max(k \cdot c_l, k \cdot c_u) \cdot (t_{end} - t_{start}) \quad (14)$$

, where c_l and c_u are the bounds of the control variable. Note that $\min(k \cdot c_l, k \cdot c_u)$ represents the minimum rate of change of $k \cdot c_{var}$ and reduces to $k \cdot c_l$ when $k > 0$. The more complicated expression is needed to preserve generality when $k < 0$. The same applies to the maximum rate of change. These linear inequalities represent the same flow tube reachability region described by equation (5) if each of the activity's control variables appear in only one continuous effect. However, note that this is not always the case. Imagine an activity *drive* with its control variable *speed*. Assume *drive* has two continuous effects that are modified by the control variable *speed*. On one hand the state variable x is modified by the flow tube $\Delta x = \text{speed} \cdot \Delta t$. On the other, the car's battery is drained according to $\Delta\text{battery} = -3 \cdot \text{speed} \cdot \Delta t$. Because the control variable *speed* that affects the battery drain is the same as the one that controls the rate of change in x, inequalities (13) and (14) can no longer represent the real state evolution of the system. These inequalities would artificially allow us to select at the same time a small value for the *speed* that drains the battery and a large one for the *speed* that makes the vehicle move fast. In these cases the reachability region needs to be represented with the original flow tube equations and the value of the control variable made a decision variable. Then the constraints become quadratic and the program is no longer linear. In its current implementation, Scotty only accepts activities in which this doesn't happen to keep the program linear but in the future, an appropriate solver will be used to handle this quadratic constraints.

According to the previous discussion, Scotty uses the following state evolution constraints for state variable x between consecutive events i and j that consider that more than one continuous effect can be operating on x simultaneously:

$$x_j \geq x_i + C_l^x \cdot (t_j - t_i) \quad (15)$$
$$x_j \leq x_i + C_u^x \cdot (t_j - t_i) \quad (16)$$
$$C_l^x = \sum_{\langle x, c_{var}, k \rangle \in A_{i \to j}^x} \min(k \cdot c_l, k \cdot c_u) \quad (17)$$
$$C_u^x = \sum_{\langle x, c_{var}, k \rangle \in A_{i \to j}^x} \max(k \cdot c_l, k \cdot c_u) \quad (18)$$
$$(19)$$

where x_k denotes the value of the state variable x at event k, t_k is the execution time of event k and $A_{i \to j}^x$ is the set of continuous effects affecting x between events i and j. Recall that c_l and c_u denote the lower and upper bounds of each state variable.

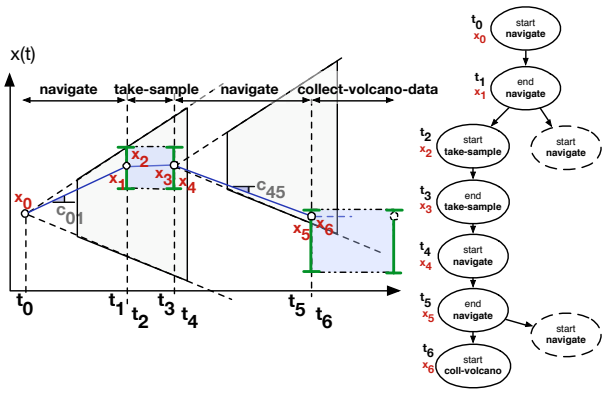


Figure 6: Flow tubes and state variable bounds for subsequent search states along the search tree. Flow tubes for the *navigate* activities define the reachable regions of x at the end of the activity (x_1, x_5). The rectangular regions show the required conditions that x needs to satisfy for the *take-sample* and *collect* activities.$x_1 = x_2 = x_3 = x_4$ and $x_5 = x_6$ because *take-sample* and *collect* don't modify x.

Note that the values of the control variables are not decision variables of the linear program. However, for each pair of consecutive events $\langle i, j \rangle$, we can compute

$$C^x_{i \to j} = \frac{x_j - x_i}{t_j - t_i} \tag{20}$$

, where $C^x_{i \to j} = \sum_{\langle x, c_{var}, k \rangle \in A^x_{i \to j}} k \cdot c_{var}$. We can use this value to find piecewise constant assignments for each control variable between events i and j. Note that there are infinite valid such possible piecewise assignments if more than one continuous effect is affecting x. This is not a problem as Scotty only needs to find one at this step. The control variable values of the example problem shown in Figure 2 were calculated in this manner.

Figure 6 shows how activities' flow tubes are handled in the linear program. The figure shows the values of state variable x as solved by the linear program at different search states. Activities' flow tubes as well as continuous conditions are shown. The consistency program is solved for each state in the search tree to determine the feasibility of the partial plan, and to extract the event times ($t_1 \dots t_6$), state variable values ($x_1 \dots x_6$), and control variables. These values keep changing as more steps are added to the plan during search. In order to find the state variable lower and upper bounds, the LP is solved twice per state variable (to minimize and maximize its value).

If the current search state is determined to be consistent, its heuristic value is computed and the state is added to the queue. If the state satisfies the goal conditions, a valid fixed plan has been found and Scotty proceeds to extract the flexible plan next. The last linear program used to extract the fixed plan tries to minimize the makespan of the plan, although a different optimization objective could be chosen.

The heuristic function used by Scotty is essentially the same used by COLIN, with minor modifications due to the use of control variables. The heuristic value for a state is the number of start or end events to reach the goal in the relaxed plan. The planning graph that COLIN expands keeps track of the state variables lower and upper bounds for each fact layer, with the caveat that activities can only grow these bounds, in a similar fashion as how Metric-FF works (Hoffmann 2003). COLIN calculates the positive gradient affecting each state variable by adding the positive rates of change of each ongoing activity (similarly with the negative gradient). In Scotty's case, these positive and negative gradients are found by adding the maximum (and minimum) rates of change given by the bounds of the control variables affecting each activity. This heuristic function is explained in detail in COLIN's journal article (Coles et al. 2012).

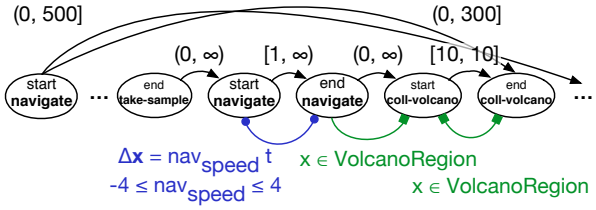


Figure 7: Example of a Qualitative State Plan that can be extracted from Scotty's solution. Start and end events are represented with labeled oval shapes. Temporal constraints are displayed in black above the events. State evolution constraints are displayed on blue. Finally, state constraints are shown in green.

Extracting a QSP

The fixed solution plan obtained in the previous section is not robust: any deviation during execution may result in an infeasible plan. In this section we discuss how we can extract a more flexible plan from the fully specified solution that can take advantage of the temporal and control flexibility of the problem.

As described in the Problem Statement, we use Qualitative State Plans (QSPs) to describe such flexible plans. A partial QSP extracted from the solution plan of the example problem is shown in Figure 7. This QSP is a temporal network of events (starts and ends of activities). These events no longer have associated precise execution times as in the fixed plan, but are connected by temporal constraints that come from the duration constraints of the activities and the problem temporal constraints (such as taking the volcano samples before the eruption finishes in 300 minutes). Start and end events of activities with continuous effects (such as *navigate*) are connected by state evolution constraints that express restrictions on how variables can vary while the activity is being executed and what the bounds of the control variables are. Finally, events can have state constraints (conditions at start or end of activities), such as being in the rendezvous region at the end of the mission, and events can be connected by state constraints, that specify the feasible tra-

jectories for the state variables between events (for example, being in the volcano region while the data is collected).

The QSP can be extracted from the fixed solution plan by traversing the fixed plan schedule and annotating the temporal constraints (activity durations), state constraints (activity conditions) and state evolution constraints (continuous effects). This requires maintaining the total order of the events so that the discrete conditions of activities hold. Lifting a partial order plan from the grounded solution or, even better, using POPF's (Coles et al. 2010) approach of planning with partial order states is possible, and is considered for future work. In that case we would also have to annotate the discrete conditions as additional constraints in the resulting QSP.

Although we could extract a flexible plan like the one defined above from the solution of any temporal planner, the fact that Scotty operates with continuous control variables makes this plan much more useful. The advantage of Scotty compared to other temporal generative planners is that Scotty can reason with continuous control variables, making this flexible plans much more useful. In effect, this gives the executive the flexibility to choose not only the execution times of the events, but also the values of the control variables that modify the continuous effects. For example, if during execution it takes the AUV longer to take the sample than initially expected, the executive will be able to increase the navigation speed of the vehicle in order to ensure that the eruption data is collected before the event ends. In short, this provides the executive with two degrees of freedom to react to disturbances: the execution times and the control variables, as long as all state transition, activity conditions and time constraints are satisfied. Algorithms exist that can execute QSPs (Hofmann and Williams 2015; 2006). These algorithms execute the plan activities online by choosing the execution times of the events and the control variables, while making sure that all constraints are propagated forward and satisfied at all times.

Empirical Evaluation

Scotty and Kongming can both reason with continuous effects modified by bounded control variables. In a previous section, we argued that Kongming's time discretization can hurt performance as the complexity of the problem increases. In Figure 8 we present a simple AUV sampling mission scenario that highlights this issue. The AUV needs to reach a certain depth range in order to take a sample. We parametrize this scenario in terms of the sampling depth that needs to be reached to take the sample. The AUV can use the action descend to modify its depth according to the bounded control variable *descent-rate*. Because Kongming discretizes time in constant time steps, increasing the target sampling depth forces Kongming to create additional fact and action layers and, additionally, more variables that the ML(N)LP solver needs to consider. As a result, the performance of Kongming degrades very fast with the target sampling depth as shown in Figure 8. While Kongming's performance degrades very fast with depth, Scotty's performance is constant (and orders of magnitude better than Kongming's). This is expected, as Scotty doesn't discretize

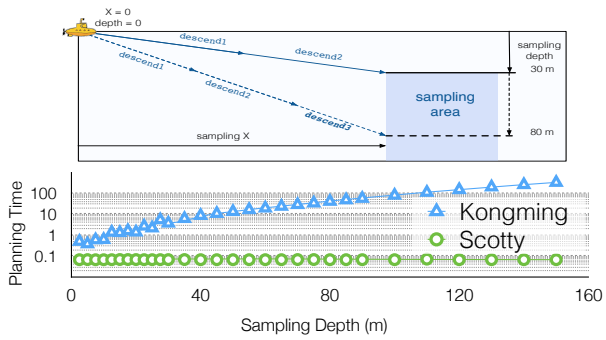


Figure 8: Sampling scenario that shows the problems of discretizing time.

	2D AUV 1	2D AUV 2	3D AUV	Firefighting 1	Firefighting 2
Kongming	3.633	9.736	13.063	1.505	20.202
Scotty	0.054	0.025	0.192	0.03	0.372

Table 3: Comparison between Kongming and Scotty in several domains. Results show planning time in seconds for each problem.

time and, therefore, is solving the same problem regardless of the depth. Table 3 shows Scotty's large performance advantage in other domains. These domains typically showcase one or more mobile robots moving in a 2D or 3D environment and trying to complete objectives that involve visiting different locations.

On the other hand COLIN/POPF are efficient heuristic forward search planners that do not present the time discretization problem. However, they are not meant for robot control and therefore do not support continuous control variables, but fixed rates of change for continuous linear time evolving effects. Let's consider an example that shows why this is not desirable for the type of problems we are interested in (Figure 9). Imagine that the AUV in the example mission needs to reach the volcano region ($x_{min} \leq x \leq x_{max}$) some prudential time after the eruption has ended so that it's safe to be nearby but before too long has passed, so

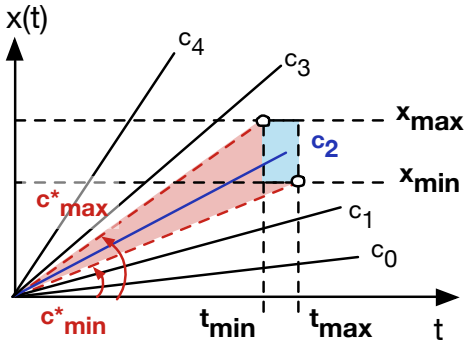


Figure 9: Discretization of rates of change.

that the collected data is still relevant ($t_{min} \leq t \leq t_{max}$). Let's consider that the robot can move with some speed (control variable) $c_{min} \leq c \leq c_{max}$. Note that in order to satisfy the constraints, the robot needs to move with some speed satisfying $c*_{min} = x_{min}/t_{max} \leq c \leq c*_{max} = x_{max}/t_{min}$. Because COLIN does not support continuous control variables, it would have to discretize the interval $[c_{min}, c_{max}]$ into a set of discrete fixed rates of change (represented as $c_0 ... c_4$ in Figure 9). The problem only becomes feasible if one of the discretized values happens to be inside the valid bounds $[c*_{min}, c*_{max}]$ (c_2 in the example). Each discretized rate of change requires a new action added to the domain, making the problem harder to solve. The required number of discretized rates of change can become arbitrarily large as different time and state constraints can modify the valid speed interval arbitrarily. This problem becomes worse if the problem contains multiple goals with similar constraints but different values that require a different discretization. If we expand this example to multiple dimensions (the robot can move in 3D space with different x, y and z velocities, for example) the problem exacerbates, as the discretization of the rate of change will need to happen simultaneously across all dimensions, which would result in a large number of discretized actions. Because Scotty supports continuous control variables, no discretization is needed and only one activity is sufficient as long as the interval given by the control variable bounds and the interval that contains the problem valid speeds have a non-empty intersection.

Finally, we compare Scotty's performance to COLIN and POPF in some of their benchmarking domains that use fixed rates of change and do not require continuous control variables. The results are shown in table 4. Some of these domains were slightly modified to substitute discrete numeric effects with fixed-rate continuous time dependent effects. The reason for that is that Scotty's current implementation doesn't support discrete numeric effects yet. This is an implementation issue and not an algorithmic problem as discrete numeric effects could be added to the planner in exactly the same way as they are handled by COLIN.

COLIN and POPF perform better in general than Scotty (about an order of magnitude) in the domains they were designed for. This is expected as they are written in C++ and their code bases have matured for years in preparation for the International Planning Competitions. On the other hand Scotty has been written from scratch in Common Lisp to better integrate with our internal software and doesn't include common optimization techniques, such as detecting compressible temporal actions, that state of the art planners do. If the goal was to develop a high performance planner with Scotty's new capabilities, integrating Scotty's advancements in COLIN's high performance code base would be the natural way to proceed.

domain	#	colin	popf	scotty
cafe delivery window	01	0.028	0.024	0.776
	02	0.014	0.056	1.098
	03	0.017	0.145	2.148
	04	0.021	0.307	3.811
	05	0.026	0.572	6.673
	06	0.039	0.963	10.540
	07	0.036	1.428	16.682
	08	0.044	2.056	25.346
	09	0.050	3.109	35.157
	10	0.060	4.171	48.323
	11	0.073	5.648	62.154
	12	0.081	7.226	86.643
linear generator	01	0.013	0.015	0.543
rovers continuous	01	0.066	0.049	1.268
	02	0.028	0.030	1.017
	03	0.096	0.125	2.319
	04	0.051	0.047	1.021
	05	-	0.701	3.486
	06	-	-	-
	07	0.324	0.538	4.219
	08	6.118	-	60.345
	09	-	-	-
	10	3.854	5.221	-
	11	1.432	264.648	-
	12	-	-	5.863

Table 4: Scotty performance results in some of COLIN's benchmark continuous domains. Table shows planning time in seconds with a timeout limit of 10 minutes. Results shown as "-" indicate that the planners couldn't solve the problem in ten minutes.

Conclusion

We have presented Scotty, a mixed discrete-continuous generative planner that fills the gap between high fidelity model predictive control approaches, that suffer from scalability problems, and temporal planners, that present an impressive performance on large problems at the cost of limited expressivity, as can only reason with behaviors with discretized control parameters. Scotty finds the middle ground between these approaches by using flow tubes to compactly encapsulate continuous effects with continuous control variables, as Kongming, and using heuristic forward search and a continuous time formulation, as COLIN. By avoiding discretization of either time or control variables, Scotty can reason with more expressive problems than COLIN and perform at least two orders of magnitude better than Kongming. Finally, Scotty produces flexible plans that are suitable for robust execution, as executives can exploit both temporal and control flexibility due to the presence of continuous control variables.

Acknowledgments

The work was partially supported by the Boeing Company under grant number MIT-BA-GTA-1. The authors would also like to thank Andrew Coles for providing a PDDL grounder tool that was used for testing our planner.

References

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1):281–300.

Cashmore, M.; Fox, M.; Larkworthy, T.; Long, D.; and Mag-

azzeni, D. 2014. AUV mission control via temporal planning. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, 6535–6541.

Coles, A.; Fox, M.; Long, D.; and Smith, A. 2008. Planning with Problems Requiring Temporal Coordination. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, 892–897.

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2009. Temporal Planning in Domains with Linear Processes. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 1671–1676.

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-Chaining Partial-Order Planning. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, 42–49.

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2012. COLIN: Planning with continuous linear numeric change. *Journal of Artificial Intelligence Research (JAIR)* 44:1–96.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence*.

Fox, M., and Long, D. 2011. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *CoRR* abs/1106.4561.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 253–302.

Hoffmann, J. 2003. The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *J Artif Intell Res(JAIR)* 20:291–341.

Hofmann, A., and Williams, B. 2006. Robust Execution of Temporally Flexible Plans for Bipedal Walking Devices. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, 386–389.

Hofmann, A. G., and Williams, B. C. 2015. Temporally and spatially flexible plan execution for dynamic hybrid systems. *Artificial Intelligence* (0 SP - EP - PY - T2 -):–.

Kautz, H. A., and Selman, B. 1999. Unifying SAT-based and Graph-based Planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, 318–325.

Léauté, T., and Williams, B. C. 2005. Coordinating Agile Systems through the Model-based Execution of Temporal Plans. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, 114–120.

Li, H. X., and Williams, B. C. 2008. Generative Planning for Hybrid Systems Based on Flow Tubes. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*, 206–213.

Li, H., and Williams, B. 2011. Hybrid Planning with Temporally Extended Goals for Sustainable Ocean Observing. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*.

Li, H. X. 2010. *Kongming: a generative planner for hybrid systems with temporally extended goals*. Ph.D. Dissertation, Massachusetts Institute of Technology.

Long, D., and Fox, M. 2003. Exploiting a Graphplan Framework in Temporal Planning. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), June 9-13, 2003, Trento, Italy*, 52–61.

Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1998. Remote Agent: to boldly go where no AI system has gone before. *Artificial Intelligence* 103(1-2):5–47.

Goal Reasoning to Coordinate Robotic Teams for Disaster Relief

Mark Roberts[1], Swaroop Vattam[1], Ronald Alford[2], Bryan Auslander[3], Tom Apker[4], Benjamin Johnson[1], and David W. Aha[4]

[1]NRC Postdoctoral Fellow; Naval Research Laboratory, Code 5514; Washington, DC
[2]ASEE Postdoctoral Fellow; Naval Research Laboratory, Code 5514; Washington, DC
[3]Knexus Research Corporation; Springfield, VA
[4]Navy Center for Applied Research in Artificial Intelligence; Naval Research Laboratory, Code 5514; Washington, DC
[1,2]*first.last*.ctr@nrl.navy.mil | [3]*first.last*@knexusresearch.com | [4]*first.last*@nrl.navy.mil

Abstract

Goal reasoning is a process by which actors deliberate to dynamically select the goals they pursue, often in response to notable events. Building on previous work, we clarify and define the Goal Reasoning Problem, which incorporates a Goal Lifecycle with refinement strategies to transition goals in the lifecycle. We show how the Goal Lifecycle can model online planning, replanning, and plan repair as instantiations of Goal Reasoning while further allowing an actor to select its goals. The Goal Reasoning Problem can be solved through goal refinement, where constraints introduced by the refinement strategies shape the solutions for successive iterations. We have developed a prototype implementation, called the Situated Decision Process, which applies goal refinement to coordinate a team of autonomous vehicles to gather information soon after a natural disaster strikes. We outline several disaster relief scenarios that progressively require more sophisticated responses. Finally, we demonstrate the prototype Situated Decision Process on the simplest of our scenarios, showing the merits of applying goal refinement to disaster relief.

1. Introduction

Robotic systems often commit to actions to achieve some goal. For example, a robot may commit to actions that attain some goal (e.g., to be at(location)) or to maintain some desirable condition (e.g., keep its battery charged). Robots also frequently act in partially-observable, dynamic environments with non-deterministic action outcomes. Consequently, robots may encounter notable events that impact their current commitments, examples of which include an exogenous event in the environment (e.g., wind disrupts vehicle navigation), a sensor reading identifies something of interest (e.g., a radio sensor reports the cell phone signal of an important person), or an executed action leads to an unanticipated outcome (e.g., a vehicle switches itself to a more urgent task, causing delay on the first task).

Robots must deliberate on their responses to notable events that impact their goals. Appropriate responses might include continuing despite the event, adjusting expectations, repairing the current plan, replanning, selecting a different goal (i.e., regoaling), deferring the original goal in favor of another goal, or dropping the goal altogether. Responses could be designed a priori or learned by the robot, but ultimately, the robot deliberates about its commitment(s) to its goal(s). *Goal Reasoning* (GR) is the capacity of an actor to deliberate about its goals, which involves formulating, prioritizing, and adjusting its goals during execution. GR actors are distinguished by their available responses, how they obtained them, and how they apply them. The degree to which an actor performs GR determines its autonomy and ability to respond to change.

We have implemented a software library for Goal Reasoning that we apply to coordinating robotic vehicle teams for Foreign Disaster Relief (FDR) operations. In the rest of the paper, we introduce FDR (§2) and our prototype system called the Situated Decision Process (SDP) (§3). We formally extend the GR Problem to online planning, demonstrating how it instantiates common systems (§4). We describe how to solve Goal Reasoning as iterative *Goal Refinement* (§5). We outline a set of FDR scenarios (§6) and detail how we applied our Goal Refinement library for the simplest of the FDR scenarios (§7). After a proof-of-concept demonstration (§8), we conclude and highlight ongoing and future work (§9). Related work is mentioned throughout the sections. The contributions of this paper over previous work (Roberts et al. 2014, 2015) include formally incorporating Goal Refinement into an online planning and execution framework (Nau 2007) and introducing an algorithm, fully outlining the set FDR scenarios, and providing richer detail of the implementation of Goal Refinement for FDR.

2. Motivating Application: Disaster Relief

We study how to coordinate a team of robotic vehicles for Foreign Disaster Relief (FDR) operations. Between the time of a tragic disaster (e.g., Typhoon Yolanda) and the arrival of support operations, emergency response personnel need information concerning the whereabouts of survivors, the condition of infrastructure, and suggested ingress and evacuation routes. Current practice for gathering this information relies heavily on humans (e.g., first responders, pilots, drone operators). A team of autonomous vehicles with sensors can facilitate such information gathering tasks, freeing humans to perform more critical tasks in FDR operations (Navy 1996). It is not tenable to tele-operate every vehicle, so we must design a system that allows humans to be "on" the control loop of vehicles without issuing every vehicle command. FDR operations present unique challenges for domain modeling because each disaster is distinct. Any system that supports FDR operations must allow personnel to tailor vehicles' tasks to the current situation. The system must also respond to notable events during execution.

An example information gathering task is shown in Figure 1 (top), which depicts a survey task for a team of fixed-wing aerial vehicles. Three vehicles (V1, V2, and V3) begin at the center base and must follow the nominal trajectories (green dashed lines) as closely as possible to maximize coverage of the areas (gray circles). The outer gray box outlines the vehicles' allowed flight envelope.

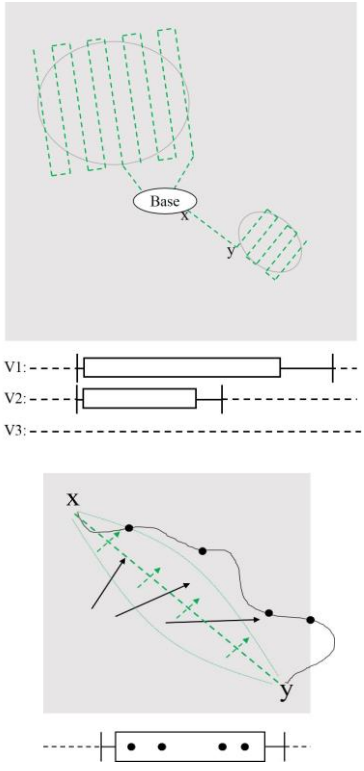


Figure 1: Examples where goal reasoning may apply in a team survey task (top) and a track following task from point x to point y (bottom) with possible notable events highlighted by dots.

Figure 1 (bottom) demonstrates notable events for a single vehicle with a next goal of at(y). The vehicle should follow the expected path (dashed line from x to y) within the preferred bounds (the curved thin green lines); staying within these bounds gives the best solution. The gray outer box is the proposed flight envelope outside of which the vehicle may negatively interact with other vehicles. The actual flight path is given by the solid arc that starts at x and ends at y. The deviating path is due to the difference between the expected wind (dashed vectors) and actual wind (solid vectors). The dots correspond to notable events that could impact the vehicle's goal commitment to be at(y). The first two points indicate where the vehicle violates the preferred trajectory while the last two points indicate the eminent and actual violation of the flight envelope.

Below each plot is a representation of the vehicle timeline(s), as described by Smith et al. (2000). The time window of the plan indicates that the plan should start executing no earlier than the earliest start time (i.e., the leftmost vertical bar) and finish by the latest finish time (the rightmost vertical bar). The large block in the middle indicates the expected duration.

3. The Situated Decision Process (SDP)

We have implemented a prototype of a system, called the Situated Decision Process (SDP), which is designed to allow flexible assignment and control of a team of robots for FDR. Figure 2 displays an abstraction of the SDP components we discuss in this paper. The SDP is partitioned into three abstract layers, each composed of components that perform specific tasks. We will briefly describe the layers and some components. A more complete exposition of the SDP and its components is provided by Roberts et al. (2015).

The *UI Layer* (colored white) manages interaction with the Operator. In this layer, the User Interface (UI) component collects operational goals and constraints from a human Operator (e.g., survey this region, look for a Very important Person (VIP) in this other region, and do not fly outside these bounds). The UI Layer conveys Operator feedback to the other components as needed and provides info to an Operator that the Operator may then decide to act on.

The *Distributed Layer* (colored black) manages the vehicles or vehicle simulation. Reactive robotic controllers often employ FSAs to determine a robot's next action. Although they are fast to execute, hand-writing FSAs is

Figure 2: An abstract view of the Situated Decision Process (SDP). Nodes are colored by layer: UI (white), Coordination (Gray), and Vehicle (black).

error prone, tedious, and brittle. Yet, creating a single robotic controller for the many FDR missions and tasks is untenable because no controller could incorporate all the necessary steps. Recent advances apply a restricted variant of Linear Temporal Logic (LTL) called General Reactivity(1) to automatically synthesize FSAs in time cubic in the size of the final FSA (Bloem et al. 2012). This layer leverages LTLMop (Kress-Gazit et al. 2009) for LTL synthesis and physicomimetics (Apker et al. 2014) to implement vehicle control.

The *Coordination Layer* (colored gray) focuses on the mission and task abstractions for the vehicle teams. Even though LTL improves the consistency and speed of FSA generation, synthesis still becomes impractical for teams in dynamic environments. Hierarchical mission planning is naturally suited to limit the FSA size for teams of vehicles (e.g., by pre-allocating missions to vehicles or by assigning vehicles to teams). Assigning specific tasks to vehicles leads to compact, manageable LTL specifications, which allows us to construct vehicle FSAs with reasonable computational effort. We employ hierarchical decomposition (task) planning because it matches well with how humans view FDR operations (Navy 1996). In particular, we apply goal refinement to coordinate those vehicle missions in support of larger FDR operations.

The Coordination and Distributed Layers of the SDP are linked via a set of *Coordination Variables*, which integrate team mission goals with the vehicle controllers by providing abstraction predicates for vehicle commands, vehicle state (e.g., current behavior and health), and abstract vehicle sensor data.

The responses of the SDP must consider relevance to the operational context. Much is unknown or dramatically different from before to the disaster. The SDP must respond appropriately to the Operator dynamically (re)allocating resources or (re)prioritizing goals as new information becomes available. The SDP should respond by confirming the Operator's intent and producing alternatives that best allocate resources to goals.

4. Goal Reasoning

Deliberating about objectives – how to prioritize and attain (or maintain) them – is a ubiquitous activity of all intentional entities (i.e., actors). For the purposes of this section, we make the simplifying assumption that an objective is a *goal*, which is a set of states the actor desires to attain or maintain. Thangarajah et al. (2011) and Harland et al. (2014) show that all goals are either attainment goals or maintenance goals, but for further simplicity we will focus almost exclusively on attainment goals in this paper. Regardless of the source, achieving goals requires deliberation on the part of the actor (e.g., a plan must be created to achieve a goal). Although our motivating application is robotic team coordination, we generally refer to any system that interleaves deliberation with acting as an *actor*, following the terminology of Ghallab et al. (2014) and Ingrand & Ghallab (2014). This section extends and clarifies early work on formalizing GR by Roberts et al. (2014).

To clarify the relationship of GR to planning, consider our adaptation of Nau's (2007) model of online planning and execution in Figure 3, which shows how a Goal Reasoner complements online planning (in black) with *Goal Reasoning* (in gray). The world is modeled as a State Transition System $\Sigma = (S, A, E, \delta)$ where: $S = \{s_0, s_1, s_2, \ldots\}$ is a set of (discrete) states that represent facts in the world; $A = \{a_1, a_2, \ldots\}$ are the actions controlled by the actor; $E = \{e_1, e_2, \ldots\}$ is a set of events not controlled by the actor; and, $\delta : S \times (A \cup E) \rightarrow 2^S$ is a state-transition function. $s_{init} \in S$ denotes the initial state of the actor. Assuming attainment goals, the actor seeks a set of transitions from s_{init} to either a single goal state $s_g \in S$ or

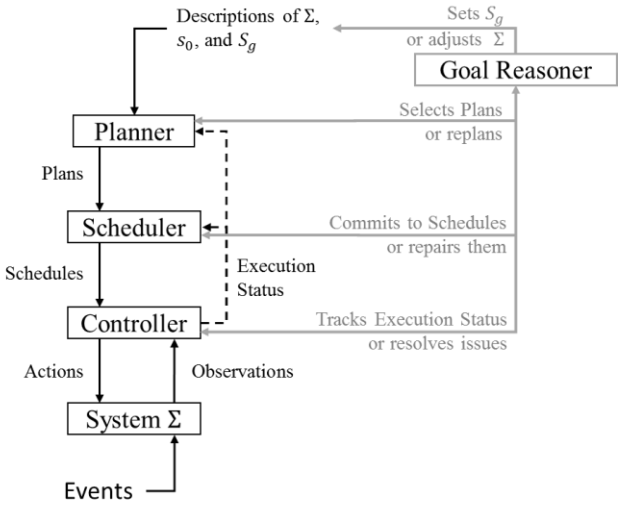


Figure 3: Incorporating Nau's (2007) Online Planning Model into a Goal Reasoning Loop.

a set of goal states $S_g \subset S$. Under classical assumptions, the planning problem can be stated: Given $\Sigma = (S, A, \delta)$, s_{init}, and a set of goal states S_g find a sequence of actions $(a_1, a_2, \ldots, a_k)$ that lead to a sequence of states $(s_{init}, s_1, s_2, \ldots, s_k)$ such that $s_1 \in \delta(s_{init}, a_1), s_2 \in \delta(s_1, a_2), \ldots, s_k \in \delta(s_{k-1}, a_k)$, and $S_g \in s_k$. However, it is rare that plans exist without some actor to execute them (cf. Pollack & Horty 1999). In *online planning*, execution status is provided to the planner as part of its deliberation. This allows the planner to adjust to dynamic events or new state in the environment. The online planning model often assumes static, external goals.

Before we define Goal Reasoning, we must clarify the notion of "state" in the GR actor, which includes *both* its external and internal state and which requires expanding the state representation beyond S. To avoid confusion with the use of the word *state* as it is typically applied in planning systems, we will use L to represent the *language* of GR. We say that the language of a GR actor is $L = L_{external} \cup L_{internal}$, where

- $L_{external}$ will often be a model of Σ but may be (or may become) a modified or incomplete version of Σ during execution or deliberation. An example of an external state for Figure 1 is at(y).
- $L_{internal}$ represents the predicates and state required for the refinement strategies (e.g., the predicates $attain(g)$ or $maintain(g)$, the state of all goals). An example of an internal state for Figure 1 is *attain*(at(y)).

We similarly extend and partition the set of goals into $L_g = External_g \cup Internal_g$. In $L_{external}$ the actor selects actions to achieve $External_g$. In $L_{internal}$ the actor selects actions to achieve $Internal_g$. Internal goals may be conditioned on external goals or vice versa. For convenience, we write goals as g and it should be clear from context whether we mean $g \in 2^S$ or $g \in 2^L$. If more context is needed we will use S_g for goals that depend on Σ and L_g for goals that depend on Z (defined next).

We model the GR actor as a State Transition System $Z = (M, R, \delta_{GR})$, where: M is the goal memory that we detail in §4.1; R is the set of refinement strategies introduced in §4.2; and $\delta_{GR} : M \times R \rightarrow M'$ is a transition function we describe in §4.3.

4.1 *The Goal Memory (M)*

The Goal Memory M stores m goals. Let g_i be the actor's i^{th} goal for $0 \leq i \leq m$. Then $N^{g_i} = \langle g_i, parent, subgoals, C, o, X, x, q \rangle$ is a goal node where:

g_i is the goal that is to be achieved (or maintained);

$parent$ is the goal whose subgoals include g_i;

$subgoals$ is a list containing any subgoals for g_i;

C is the set of constraints on g_i. Constraints could be temporal (finish by a certain time), ordering (do x before y), maintenance (remain inside this area), resource (use a specific vehicle), or computational (only use so much CPU or memory).

o is current lifecycle mode (see Figure 4 and §4.2).

X is a set of expansions that will achieve the goal. The kind of expansions for a goal depend on its type. For goals from Σ, expansions might be a plan set Π. But other goals might expand into a goal network, a task network, a set parameters for flight control, etc. The **expand** strategy, described in §4.2, creates X.

$x \in X$ is the currently selected expansion. This selection is performed with the **commit** strategy.

q is a vector of one or more quality metrics. For example, these could include the *priority* of a goal, the *inertia* of a goal indicating a bias against changing its current mode because of prior commitments, the net value (e.g., cost, value, risk, reward) associated with achieving g_i, using the currently selected expansion $x \in X$, the parallel execution time (i.e., the schedule makespan) or the number of plan steps.

The constraints will be discussed in §5, where we detail how a GR actor refines goals. A partition $C = C^{provided} \cup C^{added}$ separates constraints into those provided to the GR process independent of whatever invoked it (e.g., human operator, meta-reasoning process, coach) and those added during refinement. Top-level constraints can be pre-encoded or based on drives (e.g., (Coddington et al. 2005; Young & Hawes 2012)). Hard constraints in C must be satisfied at all times, while soft constraints should be satisfied if possible.

Our use of *goal memory* is distinct from its typical use in cognitive science, where goal memory is typically presented as a mental construct with representations and processes that are used to store and manage goal-related requirements of the task that a cognitive agent happened to be engaged in (e.g., Altmann & Trafton 1999; Anderson & Douglass 2001; Choi 2011). While issues such as interference level, strengthening, and priming constraints are key requirements to mimic human memory (Altmann & Trafton 2002), we ignore any such considerations because we are not concerned with the cognitive plausibility of our goal memory model.

4.2 Refinement strategies (R)

The actor applies a set of refinement strategies R to transition goal nodes in M. The Goal Lifecyle (Figure 4) captures the *possible* decision points of goals in the SDP. Decisions consist of applying a *strategy* (arcs in Figure 4) to transition a goal node N^g among *modes* (rounded boxes). For convenience, we sometimes refer to the goal node N^g as simply the goal g, though it should be clear

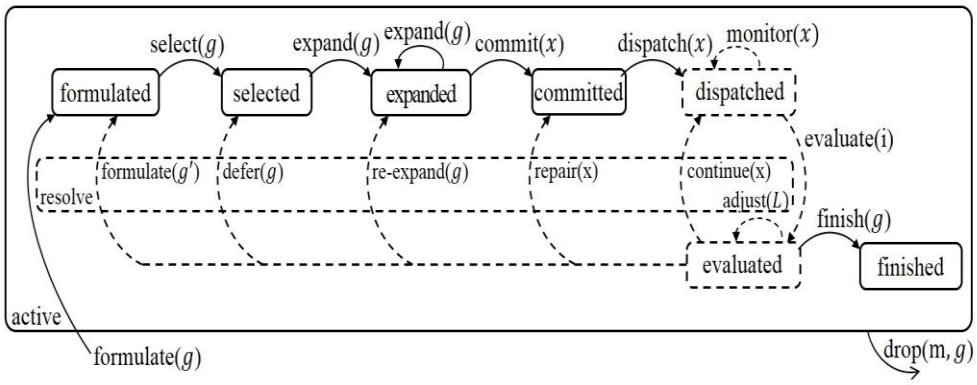


Figure 4: The Goal Lifecycle (Roberts et al. 2014). Strategies (arcs) denote possible decision points of an actor, while modes (rounded boxes) denote the status of a goal (set) in the goal memory.

that any strategies are functions that transition some N^g. In §6 we will detail strategies that we implemented for the FDR application that motivated this work.

Goal nodes in an *active* mode are those that have been formulated but not yet dropped. The **formulate** strategy determines when a new goal node is created. Vattam et al. (2013) describe goal formulation strategies. The **drop** strategy causes a goal node to be "forgotten" and can occur from any active mode; this strategy may store the node's history for future deliberation. To **select** N^g indicates intent and requires a formulated goal node. The **expand** strategy decomposes N^g into a goal network (e.g., a tree of subgoal nodes) or creates a (possibly empty) set of expansions X. Expansion is akin to the "planning" step, but is renamed here to generalize it from specific planning approaches. The **commit** strategy chooses an expansion $x \in X$ for execution; a static strategy or domain-specific quality metrics may rank possible expansions for selection. The **dispatch** strategy slates x for execution; it may further refine x prior to execution (e.g., it may allocate resources or interleave x's execution with other expansions).

Goal nodes in executing modes (Figure 4, dashed lines) can be subject to transitions resulting from expected or unexpected state changes in Σ or Z. The **monitor** strategy checks progress for N^g during execution. Execution updates, including notification that the executive has completed the tasks for the goal, arrive through the **evaluate** strategy. In a nominal execution, the information can be either resolved through a **continue** strategy after which the **finish** strategy marks the goal node as *finished*.

When notable events occur during execution, the **evaluate** strategy determines how they impact goal node execution and the **resolve** strategies define the possible responses. If the evaluation does not impact N^g, the actor can simply **continue** the execution. However, if the event impacts the current execution other strategies may apply. One obvious choice is to modify the world model (i.e., Σ or Z) using **adjust**, but adjusting its model does not resolve the mode of N^g and further refinements are required. The **repair** strategy repairs the expansion x so that it meets the new context; this is frequently called plan repair. If no repair is possible (or desired) then the **re-expand** strategy can reconsider a new plan in the revised situation for the same goal; this is frequently called replanning. The **defer** strategy postpones the goal, keeping the goal node *selected* but removing it from execution. Finally, **formulate** creates a revised goal g'; the actor then may drop the original goal g to pursue g' or it could consider both goals in parallel.

We partition $R = R^{provided} \cup R^{added} \cup R^{learned}$ to distinguish between representations that the actor was provided prior to the start of its lifetime (e.g., through design decisions), representations that were added to its model as a result of execution in an environment (e.g., a new object is sensed), and those it learned for itself (e.g., the actor adjusts its expectations for an action after experience).

4.2 *The Transition Function (δ_{GR})*

Not every strategy will apply to every goal or every situation. The transition function δ_{GR} specifies the allowed transitions between modes. In a domain-independent fashion, δ_{GR} is defined by the arcs in the lifecycle. However, a system or domain may modify (through composition, ablation, or additional constraints) the transitions for M. For example, in FDR operations, human approval is required before the SDP can commit to vehicle flight paths. In such a case, additional constraints on the commit strategy would ensure that Operator consent is obtained before a vehicle actually flies a trajectory.

4.3 *Instantiations of the Goal Reasoning Problem*

The Goal Reasoning Problem distinguishes systems by their design choices and, thus, facilitates their comparison. Figure 5 shows how different instantiations of the Goal Lifecycle can represent iterative plan repair (e.g., Chien et al. 2000), replanning (e.g., Yoon et al. 2007), and Goal-Driven Autonomy (e.g., Klenk et al. 2013).

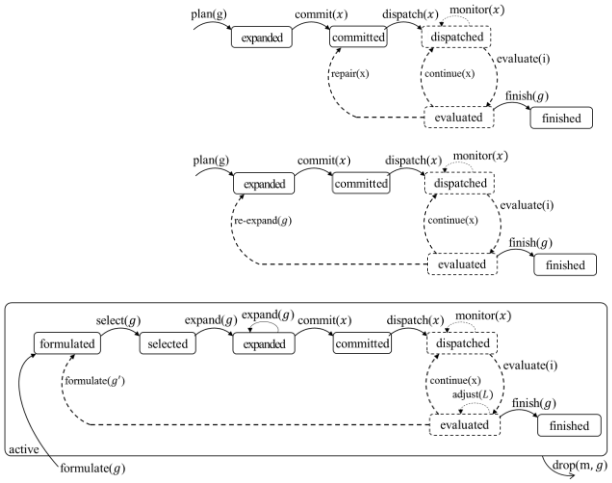


Figure 5: Instantiations of the Goal Lifecycle that incorporate plan repair (top), replanning (middle), and Goal-Driven Autonomy (bottom).

4.4 *The Goal Reasoning Problem*

We can now define the Goal Reasoning Problem. Let L_{init} be the initial state of the actor, which includes s_{init}. Then, the Goal Reasoning Problem can be stated:

> *Given* Z and L_{init}, *a GR actor examines its goal memory* M_t *at time* t *and chooses a strategy that maximizes its long-term rewards using* $\sum_t^{\infty} \gamma^t reward_t$, *where* γ^t *is a discount factor and* $reward_t: M_t \times R_t \rightarrow \mathbb{R}_t$ *yields the actor's reward of applying one or more refinement strategies at time* t.

Roberts et al. (2014) showed how this problem could be modelled as an MDP or Reinforcement Learning problem. However, we apply neither of these in our implementation.

4.5 *Related Work*

The Goal Lifecycle bears close resemblance to that of Harland et al. (2014) and earlier work (Thangarajah et al., 2010). They present a Goal Lifecycle for BDI agents, provide operational semantics for their lifecycle, and demonstrate the lifecycle on a Mars rover scenario. It remains future work to more fully characterize the overlap of their lifecycle with the Goal Lifecycle we define. Work by Winikoff et al. (2010) has also linked Linear Temporal Logic to the expression of goals. Our work differs in that it focuses on teams of robots rather than single agents.

Our approach of coordinating behaviors with constraint-based planning is inspired by much of the work mentioned by Rajan, Py, and Barriero (2013) and Myers (1999).. Our Team Executive leverages the Executive Assistant of Berry et al. (2003).

5. Goal Reasoning as Goal Refinement

Solutions to the Goal Reasoning Problem can be solved through refinement search, a process we call goal refinement. Goal refinement builds on planning as refinement search (Kambhampati 1994, 1997; Kambhampati et al. 1995). Refinement planning employs a split and prune model of search, where plans are drawn from a candidate space K. Let a search node N be a constraint set that implicitly represents a candidate set drawn from K. Refinement operators transform a node N_i at layer i into k children $\langle N_{j1}, N_{j2,}, \dots, N_{jk} \rangle$ at layer $j = i + 1$ by adding constraints that further restrict the candidate sets in the next layer. If the constraints are inconsistent then the candidate set is empty. Let $N_{\emptyset}$ represent an initial node whose candidate set equals K and results from only the initial constraint set provided in the problem description (from the perspective of the search process, the refined constraints are empty, thus the subscript $\emptyset$). The RefinePlan algorithm recursively applies refinements to add constraints until a solution is found. A desirable property of refinements is that subsequent recursive calls result in smaller candidate subsets. Thus the constraints aid search by pruning the solution space, identifying inconsistent nodes, and providing backtracking points. Instantiations of RefinePlan correspond to variants of classical planning search algorithms. Plan refinement equates different kinds of planning algorithms in plan-space and state-space planning. Extensions incorporated other forms of planning and clarify issues in the Modal Truth Criterion (Kambhampati and Nau 1994). More recent formalisms such as angelic hierarchical plans (Marthi et al. 2008) and hierarchical goal networks (Shivashankar et al. 2013) can also be viewed as leveraging plan refinement. The focus on constraints in plan refinement allows a natural extension to the many integrated planning and scheduling systems that use constraints for temporal and resource reasoning.

Figure 6 shows a Goal Refinement algorithm. Goal Refinement begins with $N_{\emptyset}^{g}$, which consists of the candidate space of all possible executions achieving g. It then applies refinement strategies from the Goal Lifecycle (see Figure 4) to N_i^g at layer i into k children $\langle N_{j1}^{g}, N_{j2,}^{g}, \dots, N_{jm}^{g} \rangle$ at layer $j = i + 1$ by modifying the goal node, which further restricts the candidate sets in the next layer. Figure 7 shows how the modes of a goal indicate successively smaller candidate sets towards eventual execution; transitions between these modes consist of adding, removing, or modifying constraints and states in N^g. Each transition increases the level of commitment the actor has made to g and increases the degree of refinement for N^g. If each refinement also reduces the candidate set of solutions, then search can be more efficient.

```
1.  RefineGoalNode (N^g, R )
2.    if mode(N^g) != active
3.          return
4.    pick a refinement operator r ∈ R
5.    refinements = apply(r,N^g)
6.    if |refinements| > 1:
7.          N^g' = nondeterministically choose a refinement
8.    else if |refinements| == 1:
9.          N^g' = choose refinement
10.   else
11.         return
12.   if N^g' is inconsistent
13.         return fail
14.   call RefineGoalNode(N^g')
```

Figure 6: A Goal Refinement algorithm

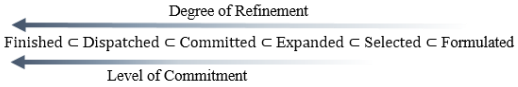


Figure 7: Modes define increasingly smaller candidate subsets for Goal Refinement

6. FDR Scenarios and GR Use Cases

Military leaders in a FDR focus on five priorities **(**Navy, 1996**)**. *Relief operations* prevent or limit further loss of life or property damage; these operations focus on identifying or deploying first responders and taking actions to provide critical sustenance and first aid. *Logistics operations* establish and maintain key areas for equipment as well as plan for the distribution of materiel and personnel to the area; these focus on determining large, medium and small landing zones for air, sea, and ground vehicles as well as determining the capacity of existing infrastructure. *Security operations* locate key personnel and maintain safety for military and civilian assets; these operations involve locating the embassy and local government personnel, determining threats to operations, and providing transportation or evacuation assistance. *Communications or information sharing operations* establish and maintain an unclassified web-based network to allow foreign planners to share information with relief organizations; these involve assessing the existing communications network and possibly supplementing it as needed. *Consequence management operations* eliminate the negative impact of intentional or inadvertent release of hazardous materials as well as potential epidemics.

Two common threads in all five priorities are *updated map data* and *reliable communications*. A team of autonomous vehicles can update the map data concerning the roads and communications network, confirm the state of any potential hazardous material or threats to operations, and provide intelligence concerning the locations of survivors. We consulted with Navy reservists who perform FDR operations to develop three scenarios that showcase how the SDP can support FDR operations. Each scenario focuses on introducing notable events or error conditions to force the SDP to respond in a coordinated way by proposing operationally relevant solutions.

Though we only discuss the scenarios in this paper, we also developed use cases based on these scenarios to independently exercise every strategy of the Goal Lifecycle. Each use case corresponds to the detection of and response to specific notable events by the Mission Manager. The use cases focus on using the **resolve** strategies of Goal Lifecycle (cf. Figure 4, dashed lines).

The vehicles in these scenarios carry three kinds of sensors. **Electro Optical (EO)** sensors that collect images. **Radio Frequency (RF)** sensors that can locate radio signals or perform radio communications. Another type of sensor detects **chemical, biological, radiation, nuclear, and explosives (CBNRE)**. We can simulate CBNRE dangers using an RF signal at a particular frequency. Alternatively, we can simulate the existence of a hazard in a mixed real-virtual environment where the vehicles are flying in the real world but sensor reports are given by a software system.

The scenarios use three vehicle types (see Figure 8). **Fixed wing Unmanned Aerial Vehicles (UAVs)** are small air vehicles such as the Bat4, Insitu ScanEagle, Unicorn or Blackjack. A UAV's operational time ranges from 2 to 20 hours, it can travel at low air speeds at altitudes up to 5000 feet, and it can carry sensor payloads up to 100kg. **Micro Aerial Vehicles (MAVs)** are small quadrotor or heptarotor UAVs such as the Acending Technologies Pelican. MAVs have operational times ranging from 3-15 *minutes*, travel close to the ground with limited range, and can carry very small EO or RF sensors. The extended scenario adds additional air and ground vehicles. **Unmanned Ground Vehicles (UGVs)** are small ground vehicles weighing about 60 pounds with a running time of 2-4 hours between charges. We use one to three iRobot Packbots PB1, PB2, and PB3. These vehicles can support significant payloads and computational power (depending on their battery life). Because MAVs are so range-limited, we also include in our scenarios **Kangaroos**, which are a combined vehicle type consisting of UGVs carrying a MAV.

These vehicles provide three atomic mission types: (1) Surveying a region with an EO sensor; (2) locate a Very Important Person (VIP) using an RF sensor; and (3) serve as a communications relay for a VIP.

Figure 8: UAV (left), UGV (middle), and MAV (right). See prose for descriptions.

6.1 *Integration Scenario*

This simple baseline scenario tests and demonstrates the major system components and their interactions; it involves a team of vehicles surveying the roads and finding a VIP in one of two Operator-selected regions. Three fixed-wing UAV vehicles fly over two pre-selected regions of interest to collect low resolution raster data in support of infrastructure assessment. When a UAV locates a survivor's cell phone signal, it circles the signal location acting as a relay until receiving further instruction.

Figure 9 demonstrates a hypothetical start of the integration scenario, when a known map is provided to the Coordination Layer and discretized to allow for sensor data collection. The Operator can specify that particular regions as likely to contain specific people. For example, areas around an embassy and airport are likely to have VIPs to the FDR operations. In this example, one VIP is located at a building on an embassy compound. The VIP's cell phone can be represented by any suitable radio signal emitter that the RF sensors on the vehicles will sense.

The Operator first identifies the specific vehicle/sensor platforms, which in this scenario includes three UAV vehicles, each with an EO and an RF sensor. The Operator then selects two regions and selects two missions (i.e., goals) for these regions: (1) VIPFound and (2) RoadsAssessed. Note that the Coordination Layer will eventually propose potential regions, as identified below in the extended scenario The Coordination Layer responds by highlighting possible trajectories over these regions, soliciting operator approval, and executing the data collection for those regions. UAV1, UAV2, and UAV3 complete a survey of these two regions of interest.

UAV1 locates the VIP cell signal and responds by switching to a VIPCommsRelayed goal. The vehicle's response is to begin hovering over the VIP signal. The UI response is to add a new avatar for the VIP (e.g., a red star) in the appropriate spot on the screen. The response in the Coordination Layer will be to **formulate** a new goal VIPCommsRelayed.

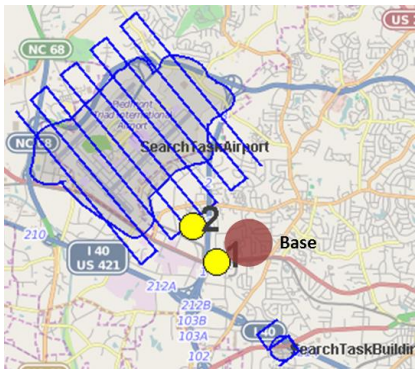


Figure 9: Example airport and VIP regions. The base is located between the regions. Also shown are the trajectories (blue lines) for two vehicles (yellow dots).

The automated formulation of this goal is central to demonstrating how GR can aid the guidance of autonomous systems. One can imagine extending this to more elaborate scenarios where the system proposes new missions for the team as missions unfold.

6.2 *Recommendation Scenario*

This scenario extends the Integration Scenario so that the SDP additionally suggests the most likely regions for finding the VIP. In this scenario, the Operator can simply accept these regions rather than have to manually highlight them. This scenario reduces Operator entry load at the start of an FDR operation; the SDP instead refines goals to obtain the map from the world model, observe probable locations of the VIP (e.g., the embassy and the airport), and suggest regions to begin exploring. To do this, the SDP analyzes known map data to create a region around an airport, a region around a building where the VIP was last spotted, and a region following the best road network between the airport and building.

6.3 *Extended Scenario*

Our most ambitious scenario involves all three vehicle types and exercises every strategy in the Goal Lifecycle. Figure 10 displays the main elements of the scenario, which takes place in a region the size of a few city blocks. Streets are named along the bottom and right side of the plot; some streets have a specific region (dotted boxes near the top and middle of the plot) where vehicles will need to survey during the scenario.

This scenario extends the Integration scenario after the point where UAV1 has identified that the VIP signal is emitted near the Embassy Compound. UAV2 and UAV3 return to base, and await further instruction. For the rest of the scenario, UAV1 circles above the embassy, tracking it and acting as a communications relay. At this point, the SDP is not aware of the Flood or Chlorine Spill because trees are blocking the flood from the UAV's camera and

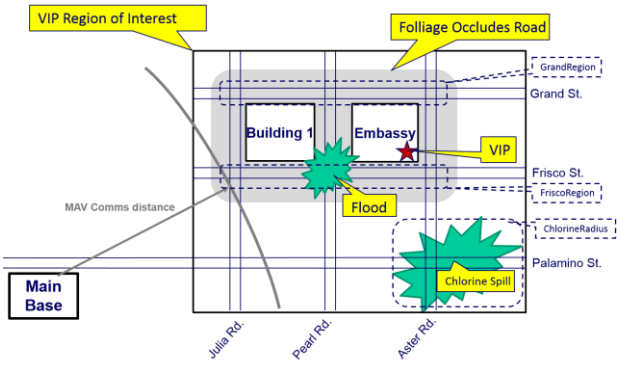


Figure 10: The Extended Scenario

the UAV is flying too high to detect the spill. The UI has already notified the operator that a VIP was found and we assume the Operator "calls" the cell phone, decides to extract this person, and adds a new goal for the autonomous team to confirm safe ingress/egress routes followed by locating the VIP within the Embassy. At the start of the scenario, the best route that can be planned to extract the VIP from the Main Base is east along Palamino St., and then north on Aster Rd., which we denote as Route A. The Coordination Layer suggests allocating two Kangaroos to explore Route A, confirms this route via the UI Layer with the human, and dispatches the approved trajectories.

During navigation of Route A, the Kangaroos detect a chlorine spill of unknown severity. The Mission Manager responds by suggesting a new allocation of three additional MAVs with CBNRE sensors to assess the extent of the spill. Alternatively, it could assign one Kangaroo to stop exploring Route A and help with this task. By now the Kangaroos should be at or near the VIP. However, because of the chlorine spill, a new route must be determined. The Mission Manager asks the Kangaroos to survey the Frisco Region (dashed lines on Frisco St.) and the flood is discovered, so Frisco St. is deemed impassable until further inquiry is done (this follow up goal should appear in the goal list of the Coordination Layer). The Mission Manager then commands the Kangaroos to survey the GrandRegion and determine Grand St. is passable. A safe route between the base and VIP is now established. The Kangaroos enter the embassy and locate the VIP. Because the FDR mission context includes locating survivors in buildings, the Mission Manager suggests the Kangaroos perform additional searching within the embassy to the Operator, who consents to this goal.

7. Goal Refinement for FDR

In this section, we detail our implementation of Goal Refinement in the SDP. We encoded the domain knowledge as a hierarchical goal network (Shivashankar et al. 2013; Geier & Bercher 2012). The SDP's goal network is currently hand-coded, but we are currently writing this model in the ANML language (Dvorák et al. 2014) and plan to integrate a full planning system in the SDP. The SDP will eventually guide vehicles in cooperation with its Operator, but in this paper we assume static mission goals and a fixed number of vehicles. We implemented the Goal Refinement model as a Java library and used this library to implement a goal network for the integration scenario. In this section, we provide details regarding how we implemented the GR actor for the Mission Manager of the SDP. We will frequently refer the goal nodes as "goals" to simplify the explanation, though it should be clear from context that these are actually the goal nodes of §4.1.

A GoalMemory class stores goal nodes in a priority queue sorted by the node priority. The priority of a node in the SDP depends on mode: formulated (10000), selected (20000), expanded (30000), committed (40000), dispatched (50000), evaluated (70000), and finished (90000). Higher priority ensures that goals further along in the lifecycle get more attention.

The Mission Manager is the GR actor in the SDP. It works with two GoalMemory objects. An InternalGoalMemory is used to initialize the system (e.g., load the domain) and store incoming information waiting to be processed. It stores any goals drawn from $L_{internal}$ (see §4). A DomainGoalMemory stores the goal nodes associated with $L_{external}$. Figure 11 displays the goal network stored in the DomainGoalMemory for the scenario in Figure 9.

Goals (i.e., goal nodes) in the SDP exist within a goal ontology. Strategies in the SDP are written as a Strategy Pattern (Gamma et al. 1994) called by the Mission Manager. Every goal in the SDP implements the StrategyInterface; goals override methods in this interface to implement their specific strategies. Unless specifically stated as overridden, goals inherit the strategies of the goals they extend. This allows default strategies to be implemented high in the goal ontology, encouraging strategy reuse.

StrategyInterface contains the following strategies: **formulate**, **select**, **expand**, **commit**, **dispatch**, **evaluate**, **resolve**, **finish**, and **drop**. This interface provides no default implementation for these strategies.

BaseGoal (***implements StrategyInterface)*** and defines the default strategies for all goals. It also implements the δ_{GR} transition function that controls which strategies can be applied; all strategies consult δ_{GR} to ensure a transition is allowed. The **formulate** strategy creates the appropriate goal node and insert the node in the correct GoalMemory. The default behavior for most strategies is to transition to the next mode if the strategy is called. However, the **expand** and **finish** strategies perform additional steps. The **expand** strategy for a goal may require interaction with a process that provides the expansions (e.g., it may require a planner). So it has a hook that allows subclasses to specify how expansions can be generated if needed. If no expansion process is provided for a goal subclass, the default behavior is to allow expand to continue. The default **finish** strategy confirms that all subgoals are finished before allowing a parent to transition to *finished*.

Figure 11: Goal decomposition during an SDP run

When the Mission Manager first loads, it places goals in each memory and adds a goal to load the domain.

> ***MaintainGoalMemory (extends BaseGoal)*** is the goal that ensures goals continue to get processed in each GoalMemory. This goal contains a queue of modified goals and notifies the MissionManager of changes. When a goal is modified, the MaintainGoalMemory goal tries to apply the next applicable strategy.
>
> ***AchieveDomainLoaded (extends BaseGoal)*** is a goal that fills the root goal in the DomainGoalMemory. It currently places the root goal in the DomainGoalMemory and finishes. However, if there were database calls or files to read for the domain to load, the **expand** strategy would be the correct strategy to implement this functionality.

Non-primitive domain goals are expanded by instantiating a sub-goal tree for the goal. The SDP commits to and dispatches the only expansion available for these goals, since this is a small example. These non-primitive goals remain in a dispatched (i.e., in-progress) state until their subgoals finish. The remaining goals are domain specific and relate to Figure 11.

> ***OperationalGoal (extends BaseGoal)*** is base type for non-primitive goals that match to the operational priorities of FDR operations. Strategies for these goals are the same as the BaseGoal.

Operational subgoals eventually decompose into AchieveTeamMission goals.

> ***AchieveTeamMission*** (***extends BaseGoal)*** modifies two strategies from the default given by BaseGoal. The **expand** strategy connects a special trajectory generator to create expansions. The trajectory generator examines many factors to create a suitable trajectory. However, the details of this are not known to the goal. The trajectory generator implements an interface that returns an expansion that is attached to the goal node after calling **expand**. The **commit** strategy requires Operator approval before it can send vehicles on a trajectory. So this strategy confirms that approval has been granted.

Expanding an AchieveTeamMission goal results in a specific VehicleMission goal, which includes details regarding a proposed allocation of a vehicle to a specific trajectory (cf. the lawnmower flight paths of Figures 1 and 9). Once the VehicleMission details are approved – either automatically or by the Operator – the Mission Manager commits and dispatches the proposed expansion of the VehicleMission for execution.

> ***VehicleMission*** (***extends BaseGoal)*** is a goal that allows the Mission Manager to track the progress toward completion of a mission. It modifies only one strategy from the default behavior. The **dispatch** strategy sets up a special listener for vehicle state that triggers the goal to move to call the evaluate strategy when a vehicle update is received. Because the default behavior of every strategy is to attempt to move to the next mode, the **evaluate** strategy will move to finished when the progress of a vehicle is above a specific threshold that indicates the task is complete.

The Coordination Manager and Team Executive then begin sending vehicle commands. The VehicleMission goal remains dispatched until new information (e.g., a progress update) causes it to become finished or need some other resolve strategy.

The Mission Manager has triggers to monitor the dispatched goals so that it will notice if the goal is stalled or completed by the executive. The Mission Manager uses a repair strategy on the original vehicle allocation to retask a vehicle for a stalled VehicleMission,

8. Demonstration

We demonstrate the SDP on the Integration Scenario (see §6.1). Figure 9 shows an airport region (upper left) and, 3-5 km away from the airport, a VIP region (lower middle) that is centered on a particular building near the suspected location of the VIP. The VIP emits a radio signal (e.g., cell phone signal). Two fixed-wing air vehicles (in yellow) are tasked with assessing the two regions and finding the VIP. They carry Electro Optical and Radio Frequency sensors that activate when the target is within their sensor radius.

The integration scenario demonstrates the SDP's key capabilities, namely that: (1) the SDP can create new goals responding to an open world (e.g., it collectively responds to the VIP being found); (2) a vehicle can make decisions autonomously (e.g., a vehicle may begin relaying the VIP once found); (3) the SDP responds to vehicles making

autonomous decisions (e.g., it notes the vehicle relaying instead of surveying when the VIP is found); and (4) the SDP can retask a vehicle to complete a mission (e.g., it retasks stalled missions to idle vehicles).

To generate 30 scenarios based on Figure 1 we select 30 random airports from OpenStreetMaps data for North Carolina (Geofabrik 2014) and then select buildings within 3-5 kilometers of the airport. Buffer regions of 300 meters around the airport and the building serve as the airport and VIP regions, respectively. Each run completes when (1) both regions are completely surveyed and the VIP is found or (2) the simulation reaches 35,000 steps. Each step is approximately one second of real time simulation. We use the MASON simulator (Luke et al. 2005) to run the scenario.

At the start of the scenario, one vehicle is assigned to assess the Aiport Region, denoted by AirportVehicle, and the other vehicle is assigned to the VIP Region, denoted VIP Vehicle. Vehicles return to the base when their fuel is sufficiently low. Vehicle behavior depends on whether the vehicles can retask themselves to relay when the VIP is found (denoted *+Relay*) or they do not relay (*–Relay*). Regardless of whether a vehicle begins relaying, the Mission Manager should always create a new "Relay VIP" goal when the VIP is found. The Mission Manager behavior depends on whether it is allowed to retask a vehicle (*+Retask*) or not (*–Retask*).

Condition 1: Find VIP (–Relay –Retask) provides a baseline. In it the vehicles detect the VIP and a new goal to relay the VIP appears when the VIP is found. Getting the SDP to do something meaningful with the "Relay VIP" goal is our next condition.

Condition 2: Relay VIP (+Relay –Retask) demonstrates that a vehicle can retask itself with a new goal by automatically relaying the VIP once found. The retasking is embedded in the Vehicle Controller (see Figure 5, line 15). However, this change of behaviors needs to be shown in the goal network, where the goal "Mission: RelayVIP" should appear after the VIP is found. However, nothing is done with the new goal and VIP Vehicle does not complete the entire survey of the VIP region because it switches its own task to relaying.

Condition 3: Relay and Retask (+Relay +Retask). To address the problem of the VIP region remaining unfinished, the Coordination Layer is allowed to retask the Airport Vehicle so it finishes the VIP Region survey after completing its area first.

When we run the simulation on the three conditions, we observe exactly the expected results. In every case, a new goal is observed in the Mission Manager after the VIP is found. In the Relay VIP condition, the VIP Vehicle begins relaying as expected, leaving the VIP Region unfinshed. When the Mission Manager is allowed to retask vehicles, we observe that all three missions complete.

This demonstration exercises most Goal Lifecycle strategies (i.e., all except **adjust** and **re-expand**). It should be clear that the nominal strategies (cf. Figure 4, solid arcs) are executed. However, it may be less clear that the demonstration also applies most of the **resolve** strategies (Figure 4, dashed lines). During the notable events of the scenario, the **evaluate** step notifies the Mission Manager of the need to deliberate. When a vehicle returns to base to recharge, the Mission Manager applies **continue** because recharging is an expected contingency behavior of surveying. When relaying is enabled (+Relay) and a vehicle switches to relaying, the Mission Manager must apply **formulate(g')** where g' is the RelayVIP goal. Then it applies **defer(Survey)** in preference to the RelayVIP goal. This leaves the Survey goal in a selected (i.e., unfinished) mode. With retasking enabled (+Retask), the Mission Manager applies **repair(Survey)** to reassign the goal to the AirportVehicle. Running the ablated versions (i.e., –Relay or –Retask) is the same thing as limiting the strategies available to the Mission Manager.

9. Summary and Future Work

We detailed our implementation of a prototype system, called the Situated Decision Process (SDP), which uses a Goal Refinement library we have constructed to coordinate teams of vehicles running LTL-synthesized FSAs in their vehicle controllers. The central contributions of this paper are clarifying the relationship of GR with Nau's (2007) model of online planning, more clearly defining the Goal Reasoning Problem and Goal Refinement, outlining a set of use cases for Foreign Disaster Relief (FDR), detailing the implementation of Goal Refinement for FDR in our prototype system and demonstrating that the SDP can respond to notable events during execution.

Future work will consist of further automating portions of the SDP, extending the demonstration to work with large teams of robotic vehicles, and enriching the domain model. For example, we plan to extend the domain model to fully encode temporal and resource concerns similar to the TREX system (Rajan, Py, and Barriero 2013). We plan to test the SDP against the more challenging scenarios with richer sensor models and higher-fidelity simulations. Ultimately, we plan to run the SDP on actual vehicles and perform user studies on its effectiveness in helping an Operator coordinate a team of vehicles in Disaster Relief.

Acknowledgements

Thanks to OSD ASD (R&E) for sponsoring this research. The views and opinions in this paper are those of the authors and should not be interpreted as representing the views or policies, expressed or implied, of NRL or OSD. We also thank the reviewers for their helpful comments.

References

Anderson, J. R., & Douglass, S. (2001). Tower of Hanoi: Evidence for the cost of goal retrieval. Journal of Experimental Psychology: Learning, Memory, and Cognition, 27, 1331–1346.

Altmann, E. M., & Trafton, J. G. (1999, August). Memory for goals: An architectural perspective. In Proc. of the 21st annual meeting of the Cognitive Science Society (Vol. 19, p. 24).

Apker, T., Liu, S.-Y., Sofge, D., and Hedrick, J.K. (2014). Application of grazing-inspired guidance laws to autonomous information gathering. Proc. of the Int'l Conference on Intelligent Robots and Systems (pp. 3828-3833). Chicago, IL: IEEE Press.

Balch, T., Dellaert, F., Feldman, A., Guillory, A., Isbell, C.L., Khan, Z., Pratt, S.C., Stein, A.N., & Wilde, H. (2006). How multirobot systems research will accelerate our understanding of social animal behavior. Proc. of the IEEE, 94(7), 1445-1463.

Berry, P., Lee, T. J., & Wilkins, D. E. (2003). Interactive execution monitoring of agent teams. Journal of Artificial Intelligence Research, 18, 217–261.

Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y. (2012). Synthesis of Reactive(1) designs. Journal of Computer and System Sciences, 78(3), 911–938.

Chien S., Knight R., Stechert A., Sherwood R., and Rabideau, G. (2000) Using Iterative Repair to Improve the Responsiveness of Planning and Scheduling. Proc. of the Conf. on Auto. Plan. and Sched.(pp. 300-307). Menlo Park, CA: AAAI.

Choi, D. (2011). Reactive goal management in a cognitive architecture. Cognitive Systems Research, 12(3), 293-308.

Coddington, A.M., Fox, M., Gough, J., Long., D., & Serina, I. (2005). MADbot: A motivated and goal directed robot. Proc. of the 20th Nat'l Conf. on Art. Intel.(pp. 1680-1681). Pittsburgh, PA: AAAI Press.

Dvorák, F., Bit-Monnot, A., Ingrand, F., & Ghallab, M. (2014). A Flexible ANML Actor and Planner in Robotics. In Working Notes, PlanRob Workshop at ICAPS. Portsmouth, NH: AAAI.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley.

Geofabrik. OpenStreetMap Data Extracts. (2014) Accessed from http://download.geofabrik.de/index.html.

Ghallab, M., Nau, D., & Traverso, P. (2014). The actor's view of automated planning and acting: A position paper. Artificial Intelligence, 208, 1–17.

Geier, T. & Bercher, P. (2011). On the decidability of HTN Planning with task insertion. In Proc. of the 22nd Int'l Joint Conf. on AI. (pp. 1955-1961). Barcelona: AAAI.

Harland, J., Morley, D., Thangarajah, J., & Yorke-Smith, N. (2014). An operational semantics for the goal life-cycle in BDI agents. Auton. Agents and Multi-Agent Systems, 28(4), 682–719.

Ingrand, F., & Ghallab, M. (2014). Robotics and artificial intelligence: A perspective on deliberation functions. AI Communications, 27(1), 63-80.

Kambhampati, S., Knoblock, C.A., & Yang, Q. (1995). Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. Art. Intelligence, 76, 168-238.

Kambhampati, S. & Nau, D. (1994). On the nature of modal truth criteria in planning. Proc. of the 12th Nat'l Conference on AI (pp. 67-97). Seattle, WA: AAAI Press.

Klenk, M., Molineaux, M., & Aha, D.W. (2013). Goal-driven autonomy for responding to unexpected events in strategy simulations. Comp. Intell., 29(2), 187-206.

Kress-Gazit, H., Fainekos, G.E., & Pappas, G.J. (2009). Temporal logic based reactive mission and motion planning. Transactions on Robotics, 25(6), 1370-1831.

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., & Balan, G. (2005). Mason: A multiagent simulation environment. Simulation, 81(7), 517-527.

Marthi, B, Russell, S., & Wolfe, J. (2008). Angelic hierarchical planning: Optimal and online algorithms. Proc. of the Int'l Conf. on Auto. Plan. & Sched. (pp. 222-231). Menlo Park, CA: AAAI.

Myers, K.L. (1999). CPEF: A continuous planning and execution framework. AI Magazine, 20(4), 63-69.

Nau, D. (2007) Current trends in Automated Planning. AI Magazine, 28(4), 43-58.

Navy, U.S. Department of. (1996) Humanitarian assistance/disaster relief operations planning, (TACMEMO 3-07.6-05). Washington, D.C.: Gov't Printing Office.

Pollack, M.E., & Horty, J. (1999). There's more to life than making plans: Plan management in dynamic, multiagent environments. AI Magazine, 20, 71-83.

Rajan, K., Py, F., & Barreiro, J. (2012). Towards deliberative control in marine robotics. In Marine Robot Autonomy (pp. 91–175). New York, NY: Springer.

Roberts, M., Vattam, S., Alford, R., Auslander, B., Karneeb, J., Molineaux, M., Apker, T., Wilson, M., McMahon, J., & Aha, D.W. (2014). Iterative goal refinement for robotics. In Working Notes of the Planning and Robotics Workshop at ICAPS. Portsmouth, NH: AAAI.

Roberts, M., Apker, T., Johnson, B., Auslander, B., Wellman, B. & Aha, D.W. (2015). Coordinating Robots for Disaster Relief. Proc. of the Conf. of the Florida AI Research Society (to appear) Hollywood, FL: AAAI.

Shivashankar, V., Alford, R., Kuter, U., & Nau, D. (2013). The GoDeL planning system: A more perfect union of domain-independent and hierarchical planning. Proc. of the 23rd Int'l Joint Conference on AI (pp. 2380-2386). Beijing, China: AAAI.

Smith, D., Frank, J., & Jonsson, A. (2000). Bridging the gap between planning and scheduling. Know. Eng. Rev., 15, 61-94.

Thangarajah, J., Harland, J., Morley, D., & Yorke-Smith, N. (2011). Operational behaviour for executing, suspending, and aborting goals in BDI agent systems. In Declarative Agent Lang. and Technologies VIII (pp. 1–21). Toronto, Canada: Springer.

Vattam, S., Klenk, M., Molineaux, M., & Aha, D. W. (2013). Breadth of approaches to goal reasoning: A research survey. In D.W. Aha, M.T. Cox, & H. Muñoz-Avila (Eds.) Goal Reasoning: Papers from the ACS Workshop (Tech. Report CS-TR-5029). College Park, MD: Univ. of Maryland, Dept. of Comp. Science.

Winikoff, M., Dastani, M., & van Riemsdijk, M. B. (2010). A unified interaction-aware goal framework. In Proc. of ECAI (pp. 1033–1034). Lisbon, Portugal: IOS Press.

Yoon, S.W., Fern, A., & Givan, R. (2007). FF-Replan: A baseline for probabilistic planning. Proc. of the 17th Int'l Conf. on Auto. Plan. and Sched. (pp. 352-359). Providence, RI: AAAI Press.

Young, J., & Hawes, N. (2012). Evolutionary learning of goal priorities in a real-time strategy game. In Proc. of the 8th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment. Stanford, CA: AAAI Press.

Planning for Serendipity - Altruism in Human-Robot Cohabitation

Tathagata Chakraborti[1] and **Gordon Briggs**[2] and **Kartik Talamadupula**[3]
Matthias Scheutz[2] and **David Smith**[4] and **Subbarao Kambhampati**[1]

Department of Computer Science[1]
Arizona State University
Tempe, AZ 85281, USA
{tchakra2,rao}@asu.edu

HRI Laboratory[2]
Tufts University
Medford, MA 02155, USA
{gbriggs,mscheutz}@cs.tufts.edu

Cognitive Learning Department[3]
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, USA
krtalamad@us.ibm.com

Intelligent Systems Division[4]
NASA Ames Research Center
Moffett Field, CA 94035, USA
david.smith@nasa.gov

Abstract

Recently there has been a lot of focus on human robot co-habitation issues that are usually orthogonal to aspects of human-robot teaming; e.g. on producing socially acceptable behaviors of robots and de-conflicting plans of robots and humans in shared environments. However, an interesting offshoot of these settings that has largely been overlooked is the problem of planning for serendipity - i.e. planning for stigmergic collaboration without explicit commitments on agents in co-habitation. In this paper we formalize this notion of planning for serendipity for the first time, and provide an Integer Programming based solution for this planning paradigm. Further, we illustrate the different modes of this planning technique on a typical Urban Search and Rescue scenario and show a real-life implementation of the ideas on the Nao Robot.

Automated planners are increasingly being used to endow robots with autonomous planning capabilities in joint human-robot task scenarios (Talamadupula et al. 2010). As the efficiency and ubiquity of planners used in these scenarios increases, so does the complexity of the various tasks that the planner can handle on behalf of the robot. Specifically, in cooperative scenarios (including human-robot teaming), the planner's role is no longer limited to only generating new plans for the robot to execute. Instead, contingent on the availability of the right information, the planner can anticipate, recognize, and further predict the future plans of other agents. Recent work (Talamadupula et al. 2014) has seen the successful deployment of this idea in scenarios where a robotic agent is trying to coordinate its plan with that of a human, and where the agents are competing for the same resource(s) and must have their plans de-conflicted in some principled manner. Indeed there has been a lot of work under the umbrella of "human-aware" planning, both in the context of path planning (Sisbot et al. 2007; Kuderer et al. 2012) and in task planning (Koeckemann, Pecora, and Karlsson 2014; Cirillo, Karlsson, and Saffiotti 2010), that aim to provide social skills to robots so as to make them produce plans conforming to desired behaviors when humans and robots operate in shared settings.

However, as we will show in this paper, the robots can be more proactive in their choices to help, and there can be different modes of collaboration (which are not necessarily aligned with just the notion of avoidance of conflicts but can involve more positive and active help) in such settings that are not predefined behavioral traits of the robots. Indeed very little attention has been paid to an important phenomenon that often occurs in the course of cooperative behavior amongst agents – *serendipity*. In this context, serendipity can be seen as the occurrence or resolution of facts in the world such that the future plan of an agent is rendered easier in some measurable sense. Note that there is no explicit team being formed here, and as such the agents do not have any commitments to help each other - this type of assistance can thus be seen as an instance of stigmergic collaboration between robots and humans in co-habitation and a way for robots (to the extent that they only exist in the setting as assistive agents to the humans) to exhibit goodwill to their human "colleagues". If the planner knows enough about the model, intentions, and state of the other agent in the scenario, it can try to *manufacture* these serendipitous circumstances. To the other agent, conditions that appear serendipitously will look remarkably similar to exogenous events (Gerevini, Saetti, and Serina 2011), and that agent may replan to take these serendipitous facts into account (thus hopefully reducing the cost of its own plan).

In this paper, we define for the first time, the notion of planning for serendipity, and outline a general framework for the different modes of such behavior. We also provide an IP-based planner that models the ideas involved in this planning paradigm. We will be using a typical USAR (Urban Search and Rescue) setting as the motivating scenario throughout the discussion to illustrate most of these ideas. Before we go into the details of the planning framework, let us first look at the scenario and understand the ideas involved in planning for serendipity in this context.

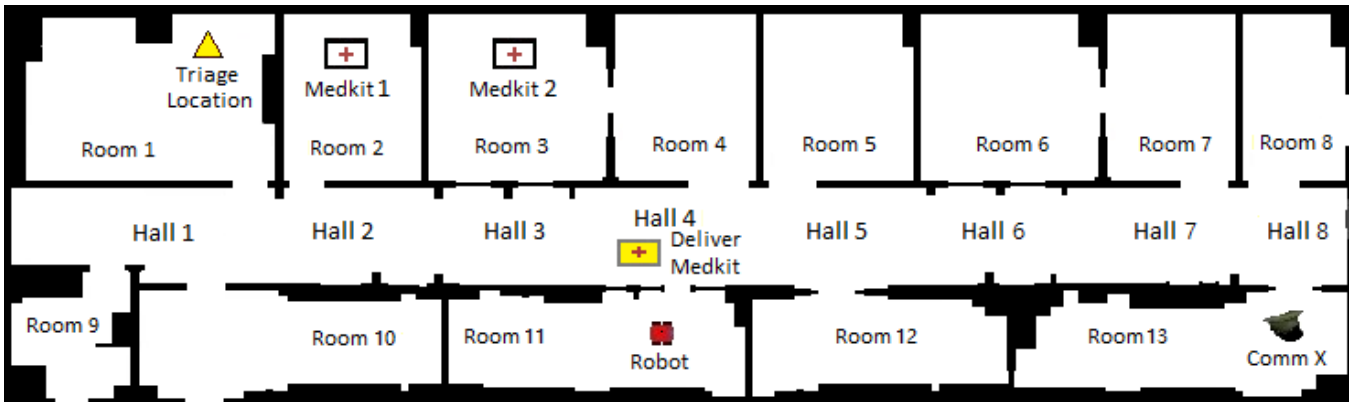


Figure 1: USAR setting involving a commander and a robot. The commanders have goal(s) to do triage at the different room locations, while the robot can help them both actively (as part of a team) or passively (planning for serendipity).

1 Overview of the USAR Scenario

Figure 1 shows a typical USAR setting, unfolding inside a building with interconnected rooms and hallways, with a human commander `CommX` and a robot. The commander has capabilities to move and conduct triage at specified locations, and he can also meet with other agents, as well as pickup, dropoff or handover medkits to accomplish their task. The robot can similarly move about, search rooms, or handover or change the position of the medkits. It can thus have its own goals (maybe from being directly assigned by the commander himself or due to long term task specifications), but can also help the commander in accomplishing his goals by fetching the medkits for him. All of these agents are autonomous agents working together or independently in the same environment, and as we discussed before, such teaming behaviors have been studied extensively in literature over the years. The specific problem we look at in this work is an interesting spin-off of such a setting - can the robot choose to help without being told to do so explicitly? What forms of assistance can this involve? Before we discuss ways to model these behaviors, we will first define a few terms related to planning in this setting.

1.1 Agent Models

Each agent α (in the current scenario α is either H or R depending on when it is being used to refer to the human or the robot respectively) in the environment is described by a domain model $D_\alpha = \langle T_\alpha, V_\alpha, S_\alpha, A_\alpha \rangle$, where T_α is a set of object types; V_α is a set of variables that describe objects that belong to T_α; S_α is a set of named first-order logical predicates over the variables V_α that describe the world state $\mathbb{W}$; and A_α is a set of operators available to the agent. The action models $a \in A_\alpha$ are represented as $a = \langle \mathbb{N}_a, \mathbb{C}_a, \mathbb{P}_a, \mathbb{E}_a \rangle$ where $\mathbb{N}_a$ denotes the name of that action; $\mathbb{C}_a$ is the cost of that action; $\mathbb{P}_a$ is the list of preconditions that must hold for the action a to be applicable; and $\mathbb{E}_a = \{eff^+(a), eff^-(a)\}$ is a list of predicates in S_α that indicates the effects of applying the action. The transition function $\delta(\cdot)$ determines the next state after the application of action a in state s as $\delta(a, s) = (s \setminus eff^-(a)) \cup eff^+(a), s \subseteq S_\alpha$. When a sequence of actions is applied to the world state, the transition function determines the resultant state by applying the actions one at a time as follows $\delta(\langle a_1, a_2, \ldots, a_n \rangle, s) = \delta(\langle a_2, \ldots, a_n \rangle, \delta(a_1, s))$.

Further, the robot can also maintain belief models Bel_α about the other agents in its environment as described in more detail in (Talamadupula et al. 2014). For the purposes of this paper, we assume that agents have complete information about the environment, and the robot has the complete domain as well as belief model of the humans. We will also assume that agents do not have communication or observation actions between themselves and the only way they can update their beliefs is through direct interaction (e.g. handing over a medkit) or exceptions in the world state while plan execution. We will relax this assumption later.

1.2 Semantics of Individual vs Composite Planning

The agents can of course, given their current state and goal, produce plans based on their own action models. However, given that the robot and the commanders are co-existing (even though independently) in the same environment with potentially mutually helpful capabilities, they could form teams or coalitions consisting of one or more agents in order to achieve a common goal.

Definition 1.0 : An *individual plan* π_α of an agent α with the domain model D_α is a mapping $\mathbb{I}_\alpha \times \mathbb{G}_\alpha \times D_\alpha \mapsto \pi_\alpha$ from the initial state $\mathbb{I}_\alpha \subseteq S_\alpha$ and the goal state $\mathbb{G}_\alpha \subseteq S_\alpha$ to an ordered sequence of actions $\pi_\alpha = \langle a_1, a_2, \ldots, a_n \rangle$, $a_i \in A_\alpha$ such that $\delta(\mathbb{I}_\alpha, \pi_\alpha) \models \mathbb{G}_\alpha$. The plan is *optimal* if whenever $\delta(\mathbb{I}_\alpha, \pi'_\alpha) \models \mathbb{G}_\alpha$, $C(\pi_\alpha) \leq C(\pi'_\alpha)$ (where $C(\pi_\alpha) = \sum_{a \in \pi_\alpha} \mathbb{C}_a$ is the cost of the plan).

Definition 1.1 : A *composite plan* $\pi_\mathbb{A}$ of a set of agents $\mathbb{A} = \{R, H\}$, referred to as the *super-agent* with a composite domain $D_\mathbb{A} = \bigcup_{\alpha \in \mathbb{A}} D_\alpha$, is defined as a mapping $\mathbb{I}_\mathbb{A} \times \mathbb{G}_\mathbb{A} \times D_\mathbb{A} \mapsto \pi_\mathbb{A}$ from the initial state $\mathbb{I}_\mathbb{A} = \bigcup_{\alpha \in \mathbb{A}} \mathbb{I}_\alpha$ and the goal state $\mathbb{G}_\mathbb{A} = \bigcup_{\alpha \in \mathbb{A}} \mathbb{G}_\alpha$ of the super-agent to an ordered sequence of *action sets* $\pi_\mathbb{A} = \langle \mu_1, \mu_2, \ldots, \mu_n \rangle$, where $\mu = \{a_1, \ldots, a_{|\mathbb{A}|}\}$, $\mu(\alpha) = a \in A_\alpha \; \forall \mu \in \mathbb{U}$

such that $\delta'(\mathbb{I}_{\mathbb{A}}, \pi_{\mathbb{A}}) \models \mathbb{G}_{\mathbb{A}}$, where the modified transition function $\delta'(\mu, s) = (s \setminus \bigcup_{a\in\mu} eff^{-}(a)) \cup \bigcup_{a\in\mu} eff^{+}(a)$. Similarly, the composite plan $\pi_{\mathbb{A}}$ is said to be optimal if whenever $\delta'(\mathbb{I}_{\mathbb{A}}, \pi'_{\mathbb{A}}) \models \mathbb{G}_{\mathbb{A}}$, $C(\pi_{\mathbb{A}}) \leq C(\pi'_{\mathbb{A}})$ (where $C(\pi_{\mathbb{A}}) = \sum_{\mu\in\pi_{\mathbb{A}}} \sum_{a\in\mu} \mathbb{C}_a$ is the cost of the plan).

A composite plan can thus be viewed as a composition of individual plans such that they together achieve a particular goal. The characteristics of the composite plan in terms of how these individual plans are composed is determined by what kind of behavior (planning for teaming or collaboration vs serendipity) we desire from the agents involved in the composite plan. Note that the contribution of the human to the composite plan is not the same as the plan he is currently executing - while generating the composite plan, the robot only ensures that this is the plan that ends up being executed (subject to different constraints discussed in detail in the Sections 2.1 and 2.2), given the human's individual plan currently "planned" for execution.

Lemma 1.1a : A composite plan $\pi_{\mathbb{A}} = \langle \mu_1, \mu_2 \ldots, \mu_T \rangle = \bigcup_{\alpha\in\mathbb{A}} \pi_\alpha$ can thus be represented as a union of plans π_α contributed by each agent $\alpha \in \mathbb{A}$ so that we can represent the original plan π_α as $\pi_{\mathbb{A}}(\alpha) = \langle a_1, a_2, \ldots, a_n \rangle,\ a_i = \mu_i(\alpha)\ \forall\ \mu_i \in \pi_{\mathbb{A}}$.

Lemma 1.1b : A composite plan $\pi_{\mathbb{A}} = \langle \mu_1, \mu_2 \ldots, \mu_T \rangle$ with $\delta'(\bigcup_{\alpha\in\mathbb{A}} \mathbb{I}_\alpha, \pi_{\mathbb{A}}) \models \mathbb{G}_{\mathbb{A}}$ guarantees that the world state $\mathbb{W} \models \mathbb{G}$ at $t = T$, if every agent $\alpha \in \mathbb{A}$ starting from the initial state $\mathbb{I}_\alpha$ at $t = 0$, executes $a_t = \mu_t(\alpha),\ \forall \mu_t \in \pi_{\mathbb{A}}$ at each time step $t \in [1, T]$. It follows that at $t < T$, $[\pi_\alpha]_{execution}$ is not necessarily same as $\langle a_1, a_2, \ldots, a_T \rangle,\ a_i = \mu_i(\alpha),\ \mu_i \in \pi_{\mathbb{A}}$.

The challenge then is to find the right composition of the individual plans into the composite plan, under constraints defined by the context. We will continue using this notion of a composite plan for the super-agent in the discussion on planning for serendipity in the next section.

2 Planning for Serendipity

In the current section we will look to formalize exactly what it means to be planning for serendipity. If the robotic agents are the ones who bring about the serendipitous moments for the human, then these moments would essentially appear as outcomes of positive exogenous events during the execution of the human's plan. Remember that, even though the humans and the robotic agents are cohabitants of the same environment, this is not a team setting, and there is no explicit commitment to help from the robots - and so the human cannot expect or plan to exploit these exogenous events. This means that, given that there are no guarantees or even expectations from the other agents, the human agent can at best only be optimal by himself. This also means that the robots, if they want to make positive interventions, must produce composite plans that are valid given the current human plan under execution. Thus it becomes incumbent on the robot to analyze the original human plan in order to determine which specific parts of the plan can be changed and which parts need to be preserved. Indeed, we will see that these notions of plan interruptibility and plan preservation are crucial to the aspect of planning for serendipity, and in the following discussion we will define the semantics of planning for serendipity in terms of plan interruptibility and plan preservation. Before we do that, however, it is worth noting at this point, that the notion of plans being enabled by positive external events is closely associated with the use of triangle table kernels (Nilsson 1985) during plan execution. However, triangle table kernels only enable positive effects internal to a plan, and cannot capture the variety of modalities in stigmergic collaboration, specifically ones that involve changes outside of the original plan under execution.

2.1 Plan Interruptibility

We start off by noting that it only makes sense to produce composite plans that have a lesser global cost than the single optimal plan of the human. However, just having a better cost does not guarantee a useful composite plan in the current context. Consider the following example. Suppose the initial positions of `medkit1` and `medkit2` are `room7` and `room3` respectively (refer to Figure 1), and `CommX` has a goal to conduct triage in `room1`. Also, suppose that the robot knows that the commander plans to pick up `medkit1` from `room7` on his way to the triage location (this being the optimal plan), while a cheaper composite plan is available if the robot chooses to pick up `medkit2` from `room3` and hands it over to `commX` in `hall4` which falls in his path. One possible way to make this happen would be to maybe lock the door to `room7` so that `commX` cannot execute his original plan any more, and switches to a plan that happens to conform to the composite optimum. However, since there is no active collaboration between the agents, in this case `CommX` might very well go looking for the keys to enter `room7`, and the serendipity is lost. Indeed, this leads us to the notion of identifying parts of the human plan as interruptible, so as to lend itself to such serendipitous execution, as follows -

Definition 2.0 : If plan $\pi_H = \langle a_1, a_2, \ldots, a_T \rangle$ of the human H with $\delta(\mathbb{I}_H, \pi_H) \models \mathbb{G}_H$, then any subplan $\pi_H^{ij} = \langle a_i, \ldots, a_j \rangle, 1 \leq i < j \leq |\pi_H|$ is *positively removable* iff $\exists \pi_{\mathbb{A}}$ for the set of agents $\mathbb{A} = \{R, H\}$ (R being the robot) such that $\delta'(\bigcup_{\alpha\in\mathbb{A}} \mathbb{I}_\alpha, \pi_{\mathbb{A}}) \models \mathbb{G}_H$ where, for some $i' > i$, $\pi_{\mathbb{A}}(H) = \big(\subseteq \pi_H[1 : i-1]\big) \bullet \pi_{\mathbb{A}}(H)[i : i'] \bullet \big(\subseteq \pi_H[j+1 : |\pi_H|]\big)$ and $C(\pi_{\mathbb{A}}(H)) < C(\pi_H)$ (here $\bullet$ means concatenation).

Definition 2.1 : A plan is *interruptible* iff it has at least one positively removable subplan.

Thus time steps $i \leq t \leq i'$ is when the (serendipitous) exceptions can occur. Note that we specify the rest of the plan to be subsequences of the original plan which ensures that the human does not need to go outside his original plan sans the part where the actual exceptions occurs.

The notion of serendipitous exceptions is closely tied to the issue of what is actually visible to the human and whether such exceptions are immediately recognizable to the human or not. While this is hard to generalize in such

non-proximal settings, one measure of this might be the length of the exception. Going back to the previous example, if the exception is just finding the locked door, then this cannot be a positive interruption because when the human goes looking for the keys then this detour is not a subplan of his original plan anymore. However, the exception can always be made to be long enough to accommodate the entire detour, but such exceptions are penalized because it is likely to be harder for the human to come up with such newer plans, partly because the entire world might not be visible to him. Note that this might mean that the formulation would sometimes prefer that the robot does not do the entire job for the human even if it were possible - this is particularly relevant to situations when the human has implicit preferences or commitments in his plan and thus shorter detours are preferable. The exact trade-off between longer interruptions (and possible interference being perceived by the human) and cheaper plans is determined by the objective function of the planner (described later in Section 2.3). We intend to do HRI studies in the future to see what kind of exceptions humans really respond to (Narayanan et al. 2015).

To account for normal human cognition, if we assume that the human replans optimally (and independently) after the serendipitous exceptions, we can modify Definition 2.0 to accommodate such adaptive behavior as follows -

Definition 2.0a : If plan $\pi_H = \langle a_1, a_2, \ldots, a_T \rangle$ of the human H with $\delta(\mathbb{I}_H, \pi_H) \models \mathbb{G}_H$, then any subplan $\pi_H^{ij} = \langle a_i, \ldots, a_j \rangle, 1 \leq i < j \leq |\pi_H|$ is *positively removable* iff $\exists \pi_{\mathbb{A}}$ for the set of agents $\mathbb{A} = \{R, H\}$ (R being the robot) such that $\delta'(\bigcup_{\alpha \in \mathbb{A}} \mathbb{I}_\alpha, \pi_{\mathbb{A}}) \models \mathbb{G}_H$ where $\pi_{\mathbb{A}}(H)[1 : i-1] \subseteq \pi_H[1 : i-1] = \langle a_1, \ldots, a_{i-1} \rangle$ and $\pi_{\mathbb{A}}(H)[i : |\pi_{\mathbb{A}}(H)|]$ is the optimal plan such that $\delta'(\delta'(\bigcup_{\alpha \in \mathbb{A}} \mathbb{I}_\alpha, \pi_{\mathbb{A}}[1 : i-1]), \pi_{\mathbb{A}}(H)[i : |\pi_{\mathbb{A}}(H)|]) \models \mathbb{G}_H$ and $C(\pi_{\mathbb{A}}(H)) < C(\pi_H)$.

We will now see what kinds of positive interruptibility accommodates serendipity for the human, and define constraints on top of Definitions 2.0-1 to determine opportunities to plan for such serendipitous moments.

2.2 Preservation Constraints

Let us now go back to the setting in Figure 1. Suppose the initial position of `medkit1` is now `room5`, and `CommX` still has a goal to conduct triage in `room1`. Clearly, one of the optimal human plans is to pick up `medkit1` from `room5` on his way to the triage location, while a cheaper composite plan is again available if the robot chooses to pick up `medkit2` from `room3` and hands it over to `commX` in `hall4` which falls in his path. However, the `CommX` does not know that the robot plans to do this, and will continue with his original plan, which makes the robot's actions redundant, and the composite plan, though cheaper, is not a feasible plan in the current setting. Specifically, since there is no expectation of interventions, the robot must preserve the plan prefix of the original plan (that appears before and independently of the intervention) in the final composite plan. This then forms the first preservation constraint -

Definition 3.0 : The composite plan $\pi_{\mathbb{A}}$ that positively removes subplan π_H^{ij} from the original plan π_H of the human is a serendipitous plan iff $\pi_{\mathbb{A}}(H)[1 : i-1] = \pi_H[1 : i-1]$, where $i = \text{argmin}_i[a = \pi_{\mathbb{A}}(H)[i] \wedge a \notin \pi_H]$, $\forall\, a \in A_H$.

Further, the composite plan must ensure that the effects of the actions of the robot R preserve the world state for the human's plan to continue executing beyond the serendipitous moment (because there is no commitment from the robot to help in future, and the human cannot plan to exploit future assistance), which provides our second preservation constraint below -

Definition 3.1 : The composite plan $\pi_{\mathbb{A}}$ that positively removes subplan π_H^{ij} from the original plan π_H of the human is a serendipitous plan iff $\delta'(\bigcup_{\alpha \in \mathbb{A}} \mathbb{I}_\alpha, \pi_{\mathbb{A}}[1 : j]) \models \delta(\mathbb{I}_H, \pi_H[1 : j])$.

We will now introduce a planner that can take into account all these constraints and produce serendipitous composite plans. Given the plan the human is executing, the robot decides on what serendipitous exceptions to introduce to make the cost of that plan lower. In doing this, the robot searches over a space of exceptions during execution time for the human's plan, as well as the length of the detours that those exceptions will cause (by "simulating" what it thinks the human will do in replanning).

2.3 The Planner

The planning problem of the robot, defined in terms of the super-agent $\mathbb{A} = \{R, H\}$ - given by $\Pi = \langle D_{\mathbb{A}}, \theta_{\mathbb{A}}, \pi_H \rangle$ - consists of the domain model $D_{\mathbb{A}}$, the problem instance $\theta_{\mathbb{A}} = \langle \mathbb{O}_{\mathbb{A}}, \mathbb{I}_{\mathbb{A}}, \mathbb{G}_{\mathbb{A}} \rangle$ (where $\mathbb{O}$ are the objects or constants in the domain, and $\mathbb{I}_{\mathbb{A}}$ and $\mathbb{G}_{\mathbb{A}}$ are the initial and goal states of the super-agent respectively) and the original plan π_H of the human. Recall that we assumed completely known belief models, which means that in our current scenario, the robot starts with the full knowledge of the human's goal(s) and can predicts the plan he is currently following (assuming optimality) - this forms π_H. Planning for serendipity involves modeling complicated constraints between the human's plan and the composite plan being generated, which is not directly suited to be handled by conventional planners. We adopt the principles for planning for serendipity outlined thus far and propose the following IP-based planner (partly following the technique for IP encoding for state space planning outlined in (Vossen et al. 1999)) to showcase these modes of behavior in our motivating scenario.

Henceforth, when we refer to the domain D_α of agent α, we will mean the grounded (with objects $\in \mathbb{O}$) version of its domain. Note that this might mean that some of the inter-agent actions (like handing over medkits) are now only available to the super-agent $\mathbb{A}$. i.e. $\bigcup_{\alpha \in \mathbb{A}} A_\alpha \subseteq A_{\mathbb{A}}$.

For the super agent, we define a binary action variable for action $a \in A_{\mathbb{A}}$ at time step t as follows:

$$x_{a,t} = \begin{cases} 1, & \text{if action a is executed by the super-agent } \mathbb{A} \\ & \text{at time step t} \\ 0, & \text{otherwise; } t \in \{1, 2, \ldots, T\} \end{cases}$$

Also, for every proposition f at step t a binary state variable is introduced as follows:

$$y_{f,t} = \begin{cases} 1, & \text{if proposition is true in plan step t} \\ 0, & \text{otherwise;} \quad \forall f \in S_{\mathbb{A}},\ t \in \{0, 1, \ldots, T\} \end{cases}$$

We also define two variables $\xi_1, \xi_2 \in [1, T], 1 \leq \xi_1 < \xi_2 \leq T$ to represent the subplan that gets positively removed by Definition 2.0. We also add a new "no-operation" action $A_\alpha \leftarrow A_\alpha \cup a_\phi \ \forall\ \alpha \in \mathbb{A}$ such that $a_\phi = \langle \mathbb{N}, \mathbb{C}, \mathbb{P}, \mathbb{E} \rangle$ where $\mathbb{N} = \texttt{NOOP}$, $\mathbb{C} = 0$, $\mathbb{P} = \{\}$ and $\mathbb{E} = \{\}$.

The IP formulation modeling the interruptibility and preservation constraints is given by (the constraints are explained after the formulation):

$$Obj\ :\ min\ \textstyle\sum_{a \in A_{\mathbb{A}}}\ \sum_{t \in \{1,2,\ldots,T\}} \mathbb{C}_a \times x_{a,t} + K\,||\xi_2 - \xi_1||$$

such that

$$y_{f,0} = 1\ \forall f \in \textstyle\bigcup_{\alpha \in \mathbb{A}} \mathbb{I}_\alpha \tag{1}$$

$$y_{f,0} = 0\ \forall f \notin \textstyle\bigcup_{\alpha \in \mathbb{A}} \mathbb{I}_\alpha \tag{2}$$

$$y_{f,T} = 1\ \forall f \in \mathbb{G}_H \tag{3}$$

$$x_{a,t} \leq y_{f,t-1}\ \forall a \in A_{\mathbb{A}},\ s.t.\ f \in \mathbb{P}_a, t \in \{1, \ldots, T\} \tag{4}$$

$$y_{f,t} \leq y_{f,t-1} + \textstyle\sum_{a \in add(f)} x_{a,t}$$
$$\text{s.t. } add(f) = \{a | f \in eff^+(a)\},\ a \in A_{\mathbb{A}},\ \forall f \in S_{\mathbb{A}},$$
$$t \in \{1, 2, \ldots, T\} \tag{5}$$

$$y_{f,t} \leq 1 - \textstyle\sum_{a \in del(f)} x_{a,t}$$
$$\text{s.t. } del(f) = \{a | f \in eff^-(a)\},\ a \in A_{\mathbb{A}}\ \forall f \in S_{\mathbb{A}},$$
$$t \in \{1, 2, \ldots, T\} \tag{6}$$

$$\xi_1 \leq \textstyle\sum_t (t \times x_{a,t})(1 - \sum_t \sum_{a \in \pi_{\alpha_1}} x_{a,t})$$
$$+ T(1 - \textstyle\sum_t x_{a,t}) + T(\sum_t \sum_{a \in \pi_{\alpha_1}} x_{a,t})$$
$$\forall a \in A_H,\ t \in \{1, 2, \ldots, T\} \tag{7a}$$

$$x_{a,t} \geq \tfrac{1}{T}(\xi_1 - t)\ \forall a \in \pi_H,\ t \in \{1, 2, \ldots, T\} \tag{7b}$$

$$x_{a,t} \leq 1 + \tfrac{1}{T}(\xi_2 - t)\ \forall a \in A_R,\ t \in \{1, \ldots, T\} \tag{8}$$

$$x_{a,t} + x_{a_\phi,t} \geq \tfrac{1}{T}(t - \xi_2)\ \forall a \in \pi_H,\ t \in \{1, 2, \ldots, T\} \tag{9}$$

$$\textstyle\sum_{a \in A_\alpha} x_{a,t} + \sum_{a \in A_{\mathbb{A}} \setminus \bigcup_{\alpha \in \mathbb{A}} A_\alpha} x_{a,t} \leq 1$$
$$\forall \alpha \in \mathbb{A},\ t \in \{1, \ldots, T\} \tag{10}$$

$$\textstyle\sum_{a \in A_{\mathbb{A}}} \sum_{t \in \{1,2,\ldots,T\}} \mathbb{C}_a \times x_{a,t} \leq cost(\pi_H) \tag{11}$$

$$\xi_1, \xi_2 \in \{1, 2, \ldots, T\},\ \xi_2 \leq \xi_1 + 1 \tag{12}$$

$$y_{f,t} \in \{0, 1\}\ \forall f \in S_{\mathbb{A}},\ t \in \{0, 1, \ldots, T\} \tag{13}$$

$$x_{a,t} \in \{0, 1\}\ \forall a \in A_{\mathbb{A}},\ t \in \{1, 2, \ldots, T\} \tag{14}$$

where K is a large constant and T is the planning horizon.

Here, the objective function minimizes the sum of the cost of the composite plan and the length of the proposed positively removable subplan. Here we assume unit cost actions, i.e. $\mathbb{C}_a = 1\ \forall a \in A_{\mathbb{A}}$ and then investigate the effect of varying the cost of the robot's actions with respect to the human's. Constraints (1) through (3) model the initial and goal conditions, while constraints (4) through (6) enforce the state equations that maintain the preconditions, and add and delete effects of the actions.

Constraint (7a) specifies the value of ξ_1 as per Definition 3.0. Specifically, $\xi_1 = \text{argmin}_i[a = \pi_{\mathbb{A}}(H)[i] \wedge a \notin \pi_H],\ \forall\ a \in A_H$. Thus $\forall a \in A_H$ we write the following inequalities (where the constraints are written in a way such that $\beta \geq T$, so as to render such cases trivial since we already have $1 \leq \xi_1 \leq T$) -

$$\xi_1 \leq \begin{cases} \beta, & \text{if } a \notin \pi_{\mathbb{A}}(H) \\ \beta, & \text{if } a \in \pi_H \\ t, & \text{if } a = \pi_{\mathbb{A}}(H)[t] \end{cases}$$

and constraint (7b) imposes Definition 3.0 as

$$x_{a,t} \begin{cases} > 0 \implies 1, & \text{if } a \in \pi_H \text{ and } t < \xi_1 \\ \in \{0, 1\}, & \text{otherwise} \end{cases}$$

Similarly, constraint (8) models Definition 3.1 by stopping actions from the robot for $t > \xi_2$ as follows -

$$x_{a,t} \begin{cases} < 1 \implies 0, & \text{if } a \in A_R \text{ and } t > \xi_2 \\ \in \{0, 1\}, & \text{otherwise} \end{cases}$$

Constraint (9) is optional and models Definition 2.0 (when ignored, Definition 2.0a is implied) as follows -

$$x_{a,t} \begin{cases} > 0 \implies 1, & \text{if } a \in \pi_H \cup a_\phi \text{ and } t > \xi_2 \\ \in \{0, 1\}, & \text{otherwise} \end{cases}$$

Constraint (10) imposes non concurrency on the actions of each agent (or inter-agent actions) during every epoch. Constraint (11) specifies that the generated composite plan should have lesser cost than the original human plan (again, this is optional). Finally constraints (12) to (14) provide the binary ranges of the variables. The constant K penalizes larger subplans from being removed (so as to minimize interference with the human's plan).

Going back to Figure 1, we note that the optimal plan for `CommX` in order to perform triage in `room1` involves picking up `medkit1` from `room2` -

```
MOVE_COMMX_ROOM13_HALL8
MOVE_REVERSE_COMMX_HALL8_HALL7
MOVE_REVERSE_COMMX_HALL7_HALL6
MOVE_REVERSE_COMMX_HALL6_HALL5
MOVE_REVERSE_COMMX_HALL5_HALL4
MOVE_REVERSE_COMMX_HALL4_HALL3
MOVE_REVERSE_COMMX_HALL3_HALL2
MOVE_REVERSE_COMMX_HALL2_ROOM2
PICK_UP_MEDKIT_COMMX_MK1_ROOM2
MOVE_COMMX_ROOM2_HALL2
MOVE_REVERSE_COMMX_HALL2_HALL1
MOVE_REVERSE_COMMX_HALL1_ROOM1
CONDUCT_TRIAGE_COMMX_ROOM1
```

However, the robot can be proactive and decide to fetch `medkit2` and hand it over to him on his way towards `room1`. Indeed, this is the plan that it produces -

```
1 - MOVE_COMMX_ROOM13_HALL8
1 - MOVE_REVERSE_ROBOT_ROOM4_ROOM3
2 - MOVE_REVERSE_COMMX_HALL8_HALL7
2 - PICK_UP_MEDKIT_ROBOT_MK2_ROOM3
3 - MOVE_REVERSE_COMMX_HALL7_HALL6
4 - MOVE_REVERSE_COMMX_HALL6_HALL5
4 - MOVE_ROBOT_ROOM3_ROOM4
5 - MOVE_REVERSE_COMMX_HALL5_HALL4
5 - MOVE_ROBOT_ROOM4_HALL4
6 - HAND_OVER_ROBOT_COMMX_MK2_HALL4
6 - HAND_OVER_ROBOT_COMMX_MK2_HALL4
8 - MOVE_REVERSE_COMMX_HALL4_HALL3
9 - MOVE_REVERSE_COMMX_HALL3_HALL2
11 - MOVE_REVERSE_COMMX_HALL2_HALL1
12 - MOVE_REVERSE_COMMX_HALL1_ROOM1
13 - CONDUCT_TRIAGE_COMMX_ROOM1
```

2.4 Planning with Communication

The dynamics of the setting change somewhat when we allow certain forms of communication to exist between the agents. Going back to the previous example, now it is no longer necessary for the robot to ensure that the prefix of the original human plan is respected (for example, the robot can inform `commX` that it is going to be in `hall4` to hand over `medkit2` to him), so that planning with communication changes the desiderata in terms of the preservation constrains in the plan generation process.

One immediate upshot of being able to communicate is that it is no longer necessary for the robot to preserve plan prefixes, and Definition 3.0 and correspondingly constraints (7b) and (7b) are no longer required. If, however, we wish to impose the interruptibility constraints from Definition 1.0 as $\pi_{\mathbb{A}}(H)[1:i-1] \subseteq \pi_H[1:i-1]$ (for a positively removable subplan π_H^{ij}) on the plan prefix, constraint $7b$ may now updated to the following -

$$x_{a,t} \leq 1 + \frac{1}{T}(t - \xi_1)$$
$$\forall a \in A_H \wedge a \notin \pi_H,\ t \in \{1, 2, \ldots, T\} \qquad (7b)$$

Finally, communication comes at a cost - too much communication might feel like interference from the point of view of the human. With this in mind, we can define the communication cost to be proportional to the number (or cost) of actions that the robot changes in the composite plan with respect to the original human plan.

Definition 3.2 : The communication cost in the composite plan $\pi_{\mathbb{A}}$ is given by $\mathcal{C} \propto \sum \mathbb{C}_a \; \forall\, a \in \pi_{\mathbb{A}}(H) \wedge a \notin \pi_H$.

Thus we update the objective function of the IP with $Obj \leftarrow Obj + \mathcal{C}$ (and remove constraints (7a) and (7b)).

Going back again to the world state in Figure 1, but now with `medkit1` in `room7`, we note that the optimal plan for `CommX` in order to perform triage in `room1` involves picking up `medkit1` from `room7`, as follows -

```
MOVE_COMMX_ROOM13_HALL8
MOVE_REVERSE_COMMX_HALL8_HALL7
MOVE_REVERSE_COMMX_HALL7_ROOM7
PICK_UP_MEDKIT_COMMX_MK1_ROOM7
MOVE_COMMX_ROOM7_HALL7
MOVE_REVERSE_COMMX_HALL7_HALL6
MOVE_REVERSE_COMMX_HALL6_HALL5
MOVE_REVERSE_COMMX_HALL5_HALL4
MOVE_REVERSE_COMMX_HALL4_HALL3
MOVE_REVERSE_COMMX_HALL3_HALL2
MOVE_REVERSE_COMMX_HALL2_HALL1
MOVE_REVERSE_COMMX_HALL1_ROOM1
CONDUCT_TRIAGE_COMMX_ROOM1
```

The plan from the previous section is no longer a valid serendipitous plan because it violates Definition 3.0, as confirmed by the planner. However, the robot can choose to communicate its intention to handover `medkit2` and indeed, the planner once again produces the plan outlined in the previous section when communication is allowed.

3 Experimental Results

In the following section we will go through simulations to illustrate some of the salient aspects of planning for serendipity, and provide a real world execution of the ideas discussed so far on the `Nao Robot`. The IP-planner has been implemented on the IP-solver `gurobi`. The planner is available at http://bit.ly/1zZvFB8. The simulations were conducted on an Intel Xeon(R) CPU E5-1620 v2 @ 3.70GHz×8 processor with a 62.9GiB memory. For the simulations, we build a suite of 200 test problems on the domain shown in Figure 1, by randomly generating positions for the two medkits and the positions of the two agents, and also randomly assigning a triage goal to the commander.

3.1 Different Flavors of Collaboration

In Table 1 we look at the full spectrum of costs incurred (to the *entire* team) by planning for individual plans to planning for serendipity (with and without communication) to optimal global plans and compare gains associated with each specific type of planning with respect to the individual optimal plans. Note that communication costs are set to zero in these evaluations so as to show the maximum gains potentially available by allowing communication. Also, for composite planning, the number of planning epochs was set to the length of the planning horizon of the original individual plan. Of course with higher planning horizons we will get more and more composite plans that make the robot do most of the work with discounted actions costs. Notice the gains in cost achieved through the different flavors of collaboration. The results also outline the expected trend of decreasing costs of the composite plan with respect to increasing discounts on the cost of the robot's actions.

Table 2 shows the effect of varying the discount factor on the percentage of problem instances that supported opportunities for the robot to plan for serendipity. That the numbers are low is not surprising given that we are planning for cases where the robot can help without being asked to, but notice how more and more instances become suitable for serendipitous collaboration as we reduce the costs incurred by the robot, indicating there is sufficient scope of exhibiting such behaviors for relatively lower costs of the robot's actions as compared to the human's.

Table 3 shows the runtime performance of the four types of planning approaches discussed above. The performance is evidently not affected much by the different modes of

Table 1: Comparison of Costs b/w w/ comm. vs w/o comm. vs composite optimal planning (as compared to average cost of 8.25 for individual plans)

Discount	w/o comm.	w/ comm.	Comp. Optimal
0%	9.82	9.72	9.70
10%	9.8115	9.6525	9.634
30%	9.7945	9.4805	9.458
50%	10.765	9.2475	9.21
70%	10.6815	8.931	8.89
90%	9.5525	8.5105	8.47

Table 2: Serendipitous Planning Opportunities w/ and w/o Comm. as a Function of the Discount Factor

Discount →	0%	10%	30%	50%	70%	90%
w/o comm.	1/200	7/200	7/200	12/200	29/200	32/200
w/ comm.	13/200	23/200	34/200	40/200	62/200	70/200

planning. Note that the time for generating the single plan is contained in these cases (for the composite plan also, the individual plan is produced to get the planning horizon).

3.2 Implementation on the Nao

We now illustrate the ideas discussed so far on the Nao Robot operating in a mock implementation of Figure 1 as shown in Figure 2. We reproduce the scenarios mentioned in Sections 2.3 and 2.4, and demonstrate how the Nao produces serendipitous moments during execution of the human's plan. A video of the demonstration is available at https://youtu.be/DvQBBOX6Qgo.

4 Conclusion

In this paper we propose a new planning paradigm - planning for serendipity - and provide a general formulation of the problem with assumptions of complete knowledge of the world state and agent models. We also illustrate how this can model proactive and helpful behaviors of autonomous agents towards humans operating in a shared setting like USAR scenarios. This of course raises interesting questions on how the approach can be adopted to a probabilistic framework for partially known models and goals of agents, and how the agents can use plan recognition techniques with observations on the world state to inform their planning process - questions we hope to address in future.

Acknowledgments

This research is supported in part by the ARO grant W911NF-13-1-0023, and the ONR grants N00014-13-1-0176, N00014-13-1-0519 and N00014-15-1-2027.

Table 3: Runtime Performance of the Planner

	w/o comm.	w/ comm.	Comp. Optimal
Time (in sec)	28.68	33.93	44.96

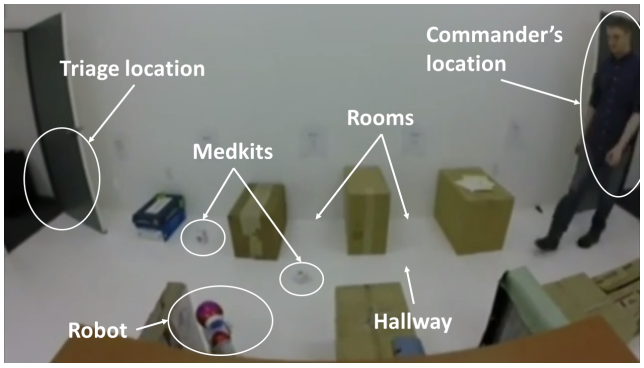


Figure 2: Demonstration of the various aspects of planning for serendipity on the Nao Robot in a mock implementation of the USAR setting from Figure 1.

References

[Cirillo, Karlsson, and Saffiotti 2010] Cirillo, M.; Karlsson, L.; and Saffiotti, A. 2010. Human-aware task planning: An application to mobile robots. *ACM Trans. Intell. Syst. Technol.* 1(2):15:1–15:26.

[Gerevini, Saetti, and Serina 2011] Gerevini, A.; Saetti, A.; and Serina, I. 2011. An approach to temporal planning and scheduling in domains with predictable exogenous events.

[Koeckemann, Pecora, and Karlsson 2014] Koeckemann, U.; Pecora, F.; and Karlsson, L. 2014. Grandpa hates robots - interaction constraints for planning in inhabited environments. In *Proc. AAAI-2010*.

[Kuderer et al. 2012] Kuderer, M.; Kretzschmar, H.; Sprunk, C.; and Burgard, W. 2012. Feature-based prediction of trajectories for socially compliant navigation. In *Proceedings of Robotics: Science and Systems*.

[Narayanan et al. 2015] Narayanan, V.; Zhang, Y.; Mendoza, N.; and Kambhampati, S. 2015. Automated planning for peer-to-peer teaming and its evaluation in remote human-robot interaction. In *Extended Abstract in ACM/IEEE International Conference on Human Robot Interaction (HRI)*.

[Nilsson 1985] Nilsson, N. 1985. Triangle tables: A proposal for a robot programming language. Technical report, Technical Note 347, AI Center, SRI International.

[Sisbot et al. 2007] Sisbot, E.; Marin-Urias, L.; Alami, R.; and Simeon, T. 2007. A human aware mobile robot motion planner. *Robotics, IEEE Transactions on* 23(5):874–883.

[Talamadupula et al. 2010] Talamadupula, K.; Benton, J.; Kambhampati, S.; Schermerhorn, P.; and Scheutz, M. 2010. Planning for human-robot teaming in open worlds. *ACM Trans. Intell. Syst. Technol.* 1(2):14:1–14:24.

[Talamadupula et al. 2014] Talamadupula, K.; Briggs, G.; Chakraborti, T.; Scheutz, M.; and Kambhampati, S. 2014. Coordination in human-robot teams using mental modeling and plan recognition. In *Intelligent Robots and Systems (IROS), IEEE/RSJ International Conference on*, 2957–2962.

[Vossen et al. 1999] Vossen, T.; Ball, M. O.; Lotem, A.; and Nau, D. S. 1999. On the use of integer programming models in ai planning. In *IJCAI*, 304–309.

Planning with Stochastic Resource Profiles: An Application to Human-Robot Co-habitation

Tathagata Chakraborti[1] and **Yu Zhang**[1] and **David Smith**[2] and **Subbarao Kambhampati**[1]

Department of Computer Science[1]
Arizona State University
Tempe, AZ 85281, USA
{tchakra2,yzhan442,rao}@asu.edu

Intelligent Systems Division[2]
NASA Ames Research Center
Moffett Field, CA 94035-1000, USA
david.smith@nasa.gov

Abstract

It is important for robotic agents to be respectful of the intentions of the human members cohabiting an environment and account for conflicts on the shared resources in the environment, in order to be acceptable members of human-robot ecosystems. In this paper we look at how maintaining predictive models of the human cohabitors in the environment can be used to inform the planning process of the robotic agents. We introduce an Integer Programming based planner as a general formulation of the "human-aware" planning problem and show how the proposed formulation can be used to model different behaviors of the robotic agent, showcasing compromise, opportunism or negotiation. Finally, we show how the proposed approach scales with the different parameters involved, and provide empirical evaluations to illustrate the pros and cons associated with the proposed style of planning.

In environments where multiple agents are working independently, but utilizing shared resources, it is important for these agents to maintain belief models of other agents so as to act intelligently and prevent conflicts. In cases where some of these agents are humans, as in assistive robots in household environments, these are required (rather than desired) capabilities of robots in order to be "socially acceptable" - this has been studied extensively under the umbrella of "human-aware" planning, both in the context of path planning (Sisbot et al. 2007; Kuderer et al. 2012) and in task planning (Cirillo, Karlsson, and Saffiotti 2009; Koeckemann, Pecora, and Karlsson 2014; Cavallo et al. 2014; Tomic, Pecora, and Saffiotti 2014). Probabilistic plan recognition can play an important role in this regard, because by not committing to a plan, that pre-assumes a particular plan for the other agent, it might be possible to minimize suboptimal (in terms of redundant or conflicting actions performed during the execution phase) behavior of the autonomous agent. Here we look at possible ways to minimize such suboptimal behavior by ways of compromise, opportunism or negotiation. There has been previous work (Beaudry, Kabanza, and Michaud 2010; Cirillo, Karlsson, and Saffiotti 2010) on some of the modeling aspects of the

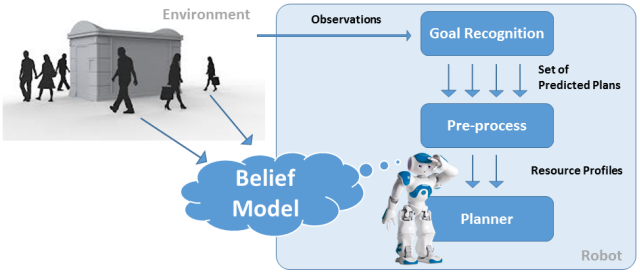


Figure 1: Architecture diagram - the robot has partial beliefs of the world, which it uses to predict and plan.

problem, in terms of planning with uncertainty in resources and constraints. In this paper we provide a unified framework of achieving these behaviors of the autonomous agents, particularly in such scenarios of human robot cohabitation.

The general framework of the problem addressed in this work is shown in Figure 1. The autonomous agent, or the robot, is acting (with independent goals) in an environment co-habited with other agents (humans), who are similarly self-interested. The robot has a model of the other agents acting independently in its environment. These models may be partial and hence the robot can only make uncertain predictions on how the world will evolve with time. However, the resources in the environment are limited and are likely to be constrained by the plans of the other agents. The robot thus needs to reason about the future states of the environment in order to make sure that its own plans do not produce conflicting states with respect to the plans of the other agents. With the involvement of humans, however, the problem is more skewed against the robot, because humans would expect a higher priority on their plans - robots that produce plans that clash with those of the humans, without any explanation, would be considered incompatible for such an ecosystem. Thus the robot will be expected to follow plans that preserve the human plans, rather than follow a globally optimal plan for itself. This aspect makes the current setting distinct from normal human robot teaming scenarios and produces a num-

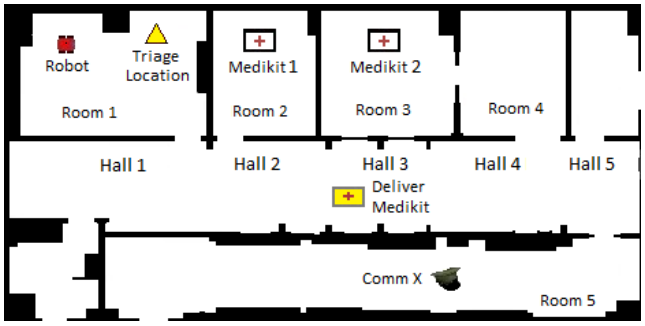


Figure 2: The running example - a human commander and a robot involved in an USAR setting, with constrained resources (medkits).

ber of its own interesting challenges. How does the robot model the humans' behavior? How does it plan to avoid friction with the human plans? If it is possible to communicate, how does it plan to negotiate and refine plans? These are the questions that we seek to address in this work. Our approach models human beliefs and defines resource profiles as abstract representations of the plans predicted on the basis of these beliefs of the human agents. The robot updates its own beliefs about the world upon receiving every new observation from its environment, and passes on the resultant profiles onto its own planner as shown in Figure 1. For the planning module, we introduce an IP-based planner that minimizes the overlap between these resource profiles and those produced by the robot's own plan in order to maintain least conflicts with the predicted human tasks in the future.

1 Planning with Resource Profiles

We will now go into details about each of the modules shown in Figure 1. We will be using a similar setting as the one described in (Talamadupula et al. 2014) (shown in Figure 2) as the running example throughout this discussion. The setting involves a commander `CommX` and a robot in a USAR (Urban Search and Rescue) scenario. The shared resources here are the two medkits - some of the plans the commander can execute will lock the use of and/or change the position of these medkits, so that from the set of probable plans of the commander we can extract a probability distribution over the usage (or even the position) of the medkit over time based on the fraction of plans that conform to these facts. These *resource availability profiles* provide a way for the agents to minimize conflicts with the other agents. Before going into details about the planner that achieves this, we will first look at how the agents are modeled and how these profiles are computed in the next section.

1.1 The Belief Modeling Component

The notion of modeling beliefs introduced by the authors in (Talamadupula et al. 2014) is adopted in this work and described briefly here. Beliefs about state are defined in terms of predicates $bel(\alpha, \phi)$, where α is an agent with belief $\phi = true$. Goals are defined by predicates $goal(\alpha, \phi)$, where agent α has a goal ϕ. The set of all beliefs that the robot ascribes to α together represents the perspective for the robot of α. This is obtained by a belief model Bel_α of agent α, defined as $\{ \phi \mid bel(\alpha, \phi) \in Bel_{self} \}$, where Bel_{self} are the first-order beliefs of the robot (e.g., $bel(self, at(self, room1))$). The set of goals ascribed to α is similarly described by $\{goal(\alpha, \phi) | goal(\alpha, \phi) \in Bel_{self}\}$.

Next, we turn our attention to the domain model D_α of the agent α that is used in the planning process. Formally, a planning problem $\Pi = \langle D_\alpha, \pi_\alpha \rangle$ consists of the domain model D_α and the problem instance π_α. The domain model of α is defined as $D_\alpha = \langle T_\alpha, V_\alpha, S_\alpha, A_\alpha \rangle$, where T_α is a set of object types; V_α is a set of variables that describe objects that belong to T_α; S_α is a set of named first-order logical predicates over the variables V_α that describe the state; and A_α is a set of operators available to the agent. The action models $a \in A_\alpha$ are represented as $a = \langle \mathbb{N}, \mathbb{C}, \mathbb{P}, \mathbb{E} \rangle$ where $\mathbb{N}$ denotes the name of that action; $\mathbb{C}$ is the cost of that action; $\mathbb{P}$ is the list of pre-conditions that must hold for the action a to be applicable; and $\mathbb{E} = \{eff^+(a), eff^-(a)\}$ is a list of predicates in S_α that indicates the effects of applying the action. The transition function $\delta(\cdot)$ determines the next state after the application of action a in state s as $\delta(a, s) = (s \setminus eff^-(a)) \cup eff^+(a), s \subseteq S_R$. For this work, we assume that the action models available to an agent are completely known to all the other agents in the scenario; that is, we rule out the possibility of beliefs on the models of other agents.

The belief model, in conjunction with beliefs about the goals / intentions of another agent, will allow the robot to instantiate a planning problem $\pi_\alpha = \langle \mathbb{O}_\alpha, \mathbb{I}_\alpha, \mathbb{G}_\alpha \rangle$, where $\mathbb{O}_\alpha$ is a set of objects of type $t \in T_\alpha$; $\mathbb{I}_\alpha$ is the *initial state* of the world, and $\mathbb{G}_\alpha$ is a set of *goals*, which are both sets of the predicates from S_α initialized with objects from $\mathbb{O}_\alpha$. First, the initial state $\mathbb{I}_\alpha$ is populated by all of the robot's initial beliefs about the agent α, i.e. $\mathbb{I}_\alpha = \{\phi \mid bel(\alpha, \phi) \in Bel_{robot}\}$. Similarly, the goal is set to $\mathbb{G}_\alpha = \{\phi \mid goal(\alpha, \phi) \in Bel_{robot}\}$. Finally, the set of objects $\mathbb{O}_\alpha$ consists of all the objects that are mentioned in either the initial state, or the goal description: $\mathbb{O}_\alpha = \{o \mid o \in (\phi \mid \phi \in (\mathbb{I}_\alpha \cup \mathbb{G}_\alpha))\}$. This planning problem instance (though not directly used in the robot's planning process) enables the goal recognition component to solve the compiled problem instances.

1.2 The Goal Recognition Component

It is unlikely for the robot to be aware of the goals of other humans in its environment completely, but it can be proactive in updating its beliefs incrementally based on observations of what the other agents are doing. To accommodate this, the robot's current belief of α's goal, $\mathbb{G}_\alpha$, is extended to a *hypothesis goal set* Ψ_α. The computation of this goal set can be done using planning graph (Blum and Furst 1995) methods. In the worst case, Ψ_α corresponds to all possible goals in the final level of the converged planning graph. Having further (domain-dependent) knowledge (e.g. in our scenario, information that `CommX` is only interested in triage-related goals) can prune some of these goals by removing the goal conditions that are not typed on the triage variable. At this point we refer to the work of Ramirez and Geffner who

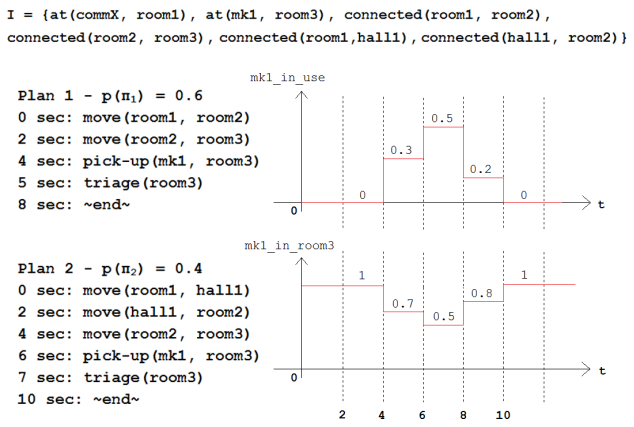


Figure 3: Different types of profiles corresponding to the two recognized plans.

in (Ramrez and Geffner 2010) provided a technique to compile the problem of goal recognition into a classical planning problem. Given a sequence of observations θ, the probability distribution Θ over $G \in \Psi_\alpha$ is recomputed by using a Bayesian update $P(G|\theta) \propto P(\theta|G)$, where the prior is approximated by the function $P(\theta|G) = 1/(1+e^{-\beta\Delta(G,\theta)})$ where $\Delta(G,\theta) = C_p(G-\theta) - C_p(G+\theta)$. Thus, solving two planning problems, with goals $G-\theta$ and $G+\theta$, gives the posterior distribution Θ over possible goals of α. We then compute the optimal plans for the goals in Ψ_α, which are used to compute the resource profiles described in the next section. Note here that one immediate advantage of using this specific goal recognition approach is that while computing the plan to a particular goal G we can reuse the compiled problem instance with the goal $G+\theta$ to ensure that the predicted plan conforms to the existing observations.

1.3 Resources and Resource Profiles

As we discussed previously, since the plans of the agents are in parallel execution, the uncertainty introduced by the commander's actions cannot be mapped directly between the commander's final state and the robot's initial state. However, given the commander's possible plans recognized by the robot, we can extract information about at what steps, or at what points of time, the shared resources in the environment are likely to be locked by the commander (given that we know what these shared resources are). This information can be represented by *resource usage profiles* that capture the expected (over all the recognized plans) variation of probability of usage or availability over time. The robot can, in turn, use this information to make sure that the profile imposed by its own plan has minimal conflicts with those of the commander's.

Formally, a profile is defined as a mapping from time step T to a real number between 0 and 1, and is represented by a set of tuples as follows $\mathcal{G} : \mathbb{N} \rightarrow [0,1] \equiv \{(t,g) : t \in \mathbb{N}, g \in [0,1],$ such that $G(t) = g$ at time step $t\}$.

The idea of the resource profiles can be handled at two levels of abstraction. Going back to our running example, shared resources that can come under conflict are the two (locatable typed objects) medkits, and the profiles over the medkits can be over both usage and location, as shown in Figure 3. These different types of profiles can be used (possibly in conjunction if needed) for different purposes. For example, just the usage profile shown on top is more helpful in identifying when to use the specific resource, while the resource when bound with the location specific groundings, as shown at the bottom can lead to more complicated higher order reasoning (e.g. the robot can decide to wait for the commander's plans to be over, as he inadvertently brings the medkit closer to it it with high probability as a result of his own plans). We will look at this again in Section 2.

Let the domain model of the robot be $D_R = \langle T_R, V_R, S_R, A_R \rangle$ with the action models $a = \langle \mathbb{N}, \mathbb{C}, \mathbb{P}, \mathbb{E} \rangle$ defined in the same way as described in Section 1.1. Also, let $\Lambda \subseteq V_R$ be the set of shared resources and for each $\lambda \in \Lambda$ we have a set of predicates $f^\lambda \subseteq S_R$ that are influenced by λ, and let $\Gamma : \Lambda \rightarrow \xi$ be a function that maps the resource variables to the set of predicates $\xi = \cup_\lambda f^\lambda$ they influence. Without any external knowledge of the environment, we can set $\Lambda = V_\alpha \cap V_R$ and $\xi = S_\alpha \cap S_R$, though in most cases these sets are much smaller. In the following discussion, we will look at how the knowledge from the hypothesis goal set can be modeled in terms of resource availability graphs for each of the constrained resources $\lambda \in \Lambda$.

Consider the set of plans Ψ_α^P containing optimal plans corresponding to each goal in the hypothesis goal set, i.e. $\Psi_\alpha^P = \{\pi_G = \langle a_1, a_2, \ldots a_t \rangle \mid G = \delta(a_t, \ldots \delta(a_2, \delta(a_1, \mathbb{I}_\alpha))) \; \forall \; G \in \Psi_\alpha$ and $a_i \in A_\alpha \forall i\}$ and let $l(\pi)$ be the likelihood of the plan π modeled on the goal likelihood distribution $\forall \; G \in \Psi_\alpha$, $p(G) \sim \Theta$ as $l(\pi_G) = c|\pi_G| \times p(G)$, where c is a normalization constant.

At each time step t, a plan $\pi \in \Psi_\alpha^P$ may lock one or more of the resources λ. Each plan thus provides a profile of usage of a resource with respect to the time step t as $\mathcal{G}_\pi^\lambda : \mathbb{N} \rightarrow \{0,1\} = \{(t,g) \mid t \in [1,|\pi|]$ and $g = 1$ if λ is locked by π at step t, 0 otherwise$\}$ such that $\mathcal{G}_\pi^\lambda(t) = g \; \forall \; (t,g) \in \mathcal{G}_\pi^\lambda$. The resultant usage profile of a resource λ due to all the plans in Ψ_α^P is obtained by summing over (weighted by the individual likelihoods) all the individual profiles as $\mathcal{G}^\lambda : \mathbb{N} \rightarrow [0,1] = \{(t,g) \mid t = 1,2,\ldots,max(|\pi|)$ and $g \propto \frac{1}{|\Psi_\alpha^P|} \sum_\pi \mathcal{G}_\pi^\lambda(t) \times l(\pi) \; \forall \; \pi \in \Psi_\alpha^P\}$.

Similarly, we can define profiles over the actual groundings of a variable (shown in the lower part of Figure 3) as $\mathcal{G}_\pi^{f^\lambda} = \{(t,g) \mid t \in [1,|\pi|]$ and $f^\lambda = 1$ at step t of plan π, 0 otherwise$\}$, and the resultant usage profile due to all the plans in Ψ_α^P is obtained as before as $\mathcal{G}^{f^\lambda} = \{(t,g) \mid t = 1,2,\ldots,max(|\pi|)$ and $g \propto \frac{1}{|\Psi_\alpha^P|} \sum_\pi \mathcal{G}_\pi^{f^\lambda}(t) \times l(\pi) \; \forall \; \pi \in \Psi_\alpha^P\}$. These profiles are helpful when actions in the robot's domain are conditioned on these variables, and the values of these variables are conditioned on the plans of the other agents in the environment currently under execution.

One important aspect of this formulation that should be noted here is that the notion of "resources" is described here in terms of the subset of the common predicates in the do-

main of the agents ($\xi \subseteq S_\alpha \cap S_R$) and can thus be used as a generalized definition to model different types of conflict between the plans between two agents. In as much as these predicates are descriptions (possibly instantiated) of the typed variables in the domain and actually refer to the physical resources in the environment that might be shared by the agents, we will stick to this nomenclature of calling them *"resources"*. We will now look at how an autonomous agent can use these resource profiles to minimize conflicts during plan execution with other agents in its environment.

1.4 Conflict Minimization

The planning problem of the robot - given by $\Pi = \langle D_R, \pi_R, \Lambda, \{G^\lambda \mid \forall \lambda \in \Lambda\}, \{G^{f^\lambda} \mid \forall f \in \Gamma(\lambda), \forall \lambda \in \Lambda\}\rangle$ - consists of the domain model D_R and the problem instance $\pi_R = \langle \mathbb{O}_R, \mathbb{I}_R, \mathbb{G}_R\rangle$ similar to that described in section 1.3, and also the constrained resources and all the profiles corresponding to them. This is because the planning process must take into account both goals of achievement as also conflict of resource usages as described by the profiles. Traditional planners provide no direct way to handle such profiles within the planning process. Note here that since the execution of the plans of the agents is occurring in parallel, the uncertainty is evolving at the time of execution, and hence the uncertainty cannot be captured from the goal states of the recognized plans alone, and consequently cannot be simply compiled away to the initial state uncertainty for the robot and solved as a conformant plan. Similarly, the problem does not directly compile into action costs in a metric planning instance because the profiles themselves are varying with time. Thus we need a planner that can handle these resource constraints that are both stochastic and non-stationary due to the uncertainty in the environment. To this end we introduce the following IP-based planner (partly following the technique for IP encoding for state space planning outlined in (Vossen et al. 1999)) as an elegant way to sum over and minimize overlaps in profiles during the plan generation process. The following formulation finds such T-step plans in case of non-durative or instantaneous actions.

For action $a \in A_R$ at step t we have an action variable:

$$x_{a,t} = \begin{cases} 1, & \text{if action a is executed in step t} \\ 0, & \text{otherwise;} \quad \forall a \in A_R,\ t \in \{1, 2, \ldots, T\} \end{cases}$$

Also, for every proposition f at step t a binary state variable is introduced as follows:

$$y_{f,t} = \begin{cases} 1, & \text{if proposition is true in plan step t} \\ 0, & \text{otherwise;} \quad \forall f \in S_R,\ t \in \{0, 1, \ldots, T\} \end{cases}$$

Note here that the plan being computed for the robot introduces a new resource consumption profile itself, and thus one optimizing criterion would be to minimize the overlap between the usage profile due to the computed plan with those established by the predicted plans of the other agents in the environment. Let us introduce a new variable to model the resource usage graph imposed by the robot as follows:

$$g_{f,t} = \begin{cases} 1, & \text{if } f \in \xi \text{ is locked at plan step } t \\ 0, & \text{otherwise;} \quad \forall f \in \xi,\ t \in \{0, 1, \ldots, T\} \end{cases}$$

For every resource $\lambda \in \Lambda$, the actions in the domain of the robot are divided into three sets - $\Omega_f^+ = \{a \in A_R \text{ such that } x_{a,t} = 1 \implies y_{f,t} = 1\}$, $\Omega_f^- = \{a \in A_R \text{ such that } x_{a,t} = 1 \implies y_{f,t} = 0\}$ and $\Omega_f^o = A_R \setminus (\Omega_f^+ \cup \Omega_f^-)$. These then specify respectively those actions in the domain that lock, free up, or do not affect the current use of a particular resource, and are used to calculate $g_{f,t}$ as part of the IP. Further, we introduce a variable $h_{f,t}$ to track preconditions required by actions in the generated plan that are conditioned on the plans of the other agents (e.g. position of the medkits are changing, and the action `pickup` is conditioned on it) as follows:

$$h_{f,t} = \begin{cases} 1, & \text{if } f \in \mathbb{P}_a \text{ and } x_{a,t+1} = 1 \\ 0, & \text{otherwise;} \quad \forall f \in \xi,\ t \in \{0, 1, \ldots, T-1\} \end{cases}$$

Then the solution to the IP should ensure that the robot only uses these resources when they are in fact most expected to be available (as obtained by maximizing the overlap between $h_{f,t}$ and G^{f^λ}). These act like *demand profiles* from the perspective of the robot.

We also add a new "no-operation" action $A_R \leftarrow A_R \cup a_\phi$ such that $a_\phi = \langle \mathbb{N}, \mathbb{C}, \mathbb{P}, \mathbb{E}\rangle$ where $\mathbb{N} =$ `NOOP`, $\mathbb{C} = 0$, $\mathbb{P} = \{\}$ and $\mathbb{E} = \{\}$.

The IP formulation is given by:

$$\begin{aligned} &min\ k_1 \textstyle\sum_{a \in A_R} \sum_{t \in \{1,2,\ldots,T\}} \mathbb{C}_a \times x_{a,t} \\ &+k_2 \textstyle\sum_{\lambda \in \Lambda} \sum_{f \in \Gamma(\lambda)} \sum_{t \in \{1,2,\ldots,T\}} g_{f,t} \times G^\lambda(t) \\ &-k_3 \textstyle\sum_{\lambda \in \Lambda} \sum_{f \in \Gamma(\lambda)} \sum_{t \in \{0,1,\ldots,T-1\}} h_{f,t} \times G^{f^\lambda}(t) \end{aligned}$$

such that

$$y_{f,0} = 1\ \forall f \in \mathbb{I}_R \setminus \xi \tag{1}$$

$$y_{f,0} = 0\ \forall f \notin \mathbb{I}_R \text{ or } f \in \xi \tag{2}$$

$$y_{f,T} = 1\ \forall f \in \mathbb{G}_R \tag{3}$$

$$x_{a,t} \leq y_{f,t-1}\ \forall a\ s.t.\ f \in \mathbb{P}_a, \forall f \notin \xi, t \in \{1, \ldots, T\} \tag{4}$$

$$h_{f,t-1} = x_{f,t}\ \forall a\ s.t.\ f \in \mathbb{P}_a, \forall f \in \xi, t \in \{1, \ldots, T\} \tag{5}$$

$$\begin{aligned} &y_{f,t} \leq y_{f,t-1} + \textstyle\sum_{a \in add(f)} x_{a,t} \\ &\text{s.t. } add(f) = \{a | f \in eff^+(a)\}, \forall f, t \in \{1, \ldots, T\} \end{aligned} \tag{6}$$

$$\begin{aligned} &y_{f,t} \leq 1 - \textstyle\sum_{a \in del(f)} x_{a,t} \\ &\text{s.t. } del(f) = \{a | f \in eff^-(a)\}, \forall f, t \in \{1, \ldots, T\} \end{aligned} \tag{7}$$

$$\textstyle\sum_{a \in A_R} x_{a,t} = 1, t \in \{1, 2, \ldots, T\} \tag{8}$$

$$\textstyle\sum_{a \in \Omega_f^+} \sum_t x_{a,t} \leq 1\ \forall f \in \xi, t \in \{1, 2, \ldots, T\} \tag{9}$$

$$\begin{aligned} &g_{f,t} = \textstyle\sum_{a \in \Omega_f^+} x_{a,t} \\ &+(1 - \textstyle\sum_{a \in \Omega_f^+} x_{a,t} - \sum_{a \in \Omega_f^-} x_{a,t}) \times g_{f,t-1} \\ &\forall f \in \xi, t \in \{1, \ldots, T\} \end{aligned} \tag{10}$$

$$h_{f,t} \times G^{f^\lambda}(t) \geq \epsilon\ \forall f \in \xi, t \in \{1, \ldots, T-1\} \tag{11}$$

$$y_{f,t} \in \{0,1\}\ \forall f \in S_R, t \in \{0,1,\ldots,T\} \quad (12)$$

$$x_{a,t} \in \{0,1\}\ \forall a \in A_R, t \in \{1,2,\ldots,T\} \quad (13)$$

$$g_{f,t} \in \{0,1\}\ \forall f \in S_R, t \in \{0,1,\ldots,T\} \quad (14)$$

$$h_{f,t} \in \{0,1\}\ \forall f \in S_R, t \in \{0,1,\ldots,T-1\} \quad (15)$$

where k_1, k_2, k_3 are constants (set manually) that determine the relative importance of each of the optimization criteria and ϵ is a small constant.

Here, the objective function minimizes the sum of the cost of the plan and the overlap between the cumulative resource usage profiles of the predicted plans and that imposed by the current plan of the robot itself while maximizing the validity of the demand profiles. Constraints (1) through (3) model the initial and goal conditions, while the value of the constrained variables are kept uninitialized (and are determined by their profiles). Constraints (4) and (5), depending on the particular predicate, enforces the preconditions, or produces the demand profiles respectively, while (6) and (7) enforces the state equations that maintain the add and delete effects of the actions. Constraint (8) imposes non concurrency on the actions, and (9) ensures that the robot does not repeat the same action indefinitely to increase its utility. Constraint (10) generates the resource profile of the current plan, while (11) maintains that actions are only executed if there is at least a small probability ϵ of success. Finally (12) to (15) provide the binary ranges of the variables.

2 Modulating the Behavior of the Robot

The IP-planner has been implemented on the IP-solver `guorbi` and integrates Ramirez *et. al.* (Ramrez and Geffner 2010) and `fast-downward` (Helmert 2011) respectively for goal recognition and plan prediction for the recognized goals. We will now go through a simplified use case, and illustrate how the resource profiles can be used to produce different behaviors of the robot by appropriately configuring the objective function and the length of the planning horizon of the IP formulation.

2.1 Compromise vs Opportunism

Let us look back at the setting in Figure 2. Consider that the robot recognizes that the goal of the commander is to perform triage in `room1`, computes his optimal plan (which ends up using `medkit1` at time steps 7 through 12) and updates the resource profiles accordingly. If now, it has its own goal to perform triage in `hall3`, the plan that it comes up with given a 12 step lookahead is shown below. Notice that the robot now opts to use the other medkit (`medkit2` in `room3`) even though its plan now incurs a higher cost in terms of execution. The robot thus can adopt a policy of *compromise* if it is possible for it to preserve the commander's (expected) plan.

```
01 - MOVE_ROBOT_ROOM1_HALL1
02 - MOVE_ROBOT_HALL1_HALL2
03 - MOVE_ROBOT_HALL2_HALL3
04 - MOVE_ROBOT_HALL3_HALL4
05 - MOVE_REVERSE_ROBOT_HALL4_ROOM4
06 - MOVE_REVERSE_ROBOT_ROOM4_ROOM3
07 - PICK_UP_MEDKIT_ROBOT_MK2_ROOM3
08 - MOVE_ROBOT_ROOM3_ROOM4
09 - MOVE_ROBOT_ROOM4_HALL4
10 - MOVE_REVERSE_ROBOT_HALL4_HALL3
11 - CONDUCT_TRIAGE_ROBOT_HALL3
12 - DROP_OFF_ROBOT_MK2_HALL3
```

Notice, however, that the commander is actually bringing the medkit to `room1` as predicted by the robot, and this is a favorable change in the world, because robot can use this `medkit` once the commander is done and incur a much lower cost of achieving its goal. The robot, indeed, realizes this once we give it a bigger time horizon to plan with, as shown below. Thus, in this case, the robot shows *opportunism* based on how it believes the world state will change.

```
01 - NOOP
02 - NOOP
03 - NOOP
   ...
12 - NOOP
13 - NOOP
14 - PICK_UP_MEDKIT_ROBOT_MK1_ROOM1
15 - MOVE_ROBOT_ROOM1_HALL1
16 - MOVE_ROBOT_HALL1_HALL2
17 - MOVE_ROBOT_HALL2_HALL3
18 - CONDUCT_TRIAGE_ROBOT_HALL3
19 - DROP_OFF_ROBOT_MK1_HALL3
```

2.2 Negotiation

In many cases, the robot will have to eventually produce plans that will have potential points of conflict with the expected plans of the commander. This occurs when there is no feasible plan with zero overlap between profiles (specifically $\sum g_{f,t} \times G^{\lambda}(t) = 0$) or if the alternative plans for the robot are too costly (as determined by the objective function). If, however, the robot is equipped with the ability to communicate with the human, then it can *negotiate* a plan that suits both. To this end, we introduce a new variable $H^{\lambda}(t)$ and update the IP as follows:

$$\begin{aligned} &min\ k_1 \textstyle\sum_{a \in A_R} \sum_{t \in \{1,2,\ldots,T\}} \mathbb{C}_a \times x_{a,t} \\ &+k_2 \textstyle\sum_{\lambda \in \lambda} \sum_{f \in \Gamma^{-1}(\lambda)} \sum_{t \in \{1,2,\ldots,T\}} g_{f,t} \times H^{\lambda}(t) \\ &-k_3 \textstyle\sum_{\lambda \in \Lambda} \sum_{f \in \Gamma^{-1}(\lambda)} \sum_{t \in \{0,1,\ldots,T-1\}} h_{f,t} \times G^{f^{\lambda}}(t) \\ &+k_4 \textstyle\sum_{\lambda \in \Lambda} \sum_{t \in \{0,1,\ldots,T\}} ||G^{\lambda}(t) - H^{\lambda}(t)|| \end{aligned}$$

$$y_{f,T} \geq h_{f,t-1}\ \forall\ a\ s.t.\ f \in \mathbb{P}_a, \forall f \in \xi, t \in \{1,\ldots,T\} \quad (5a)$$

$$H^{\lambda}(t) \in [0,1]\ \forall \lambda \in \Lambda, t \in \{0,1,\ldots,T\} \quad (16)$$

$$H^{\lambda}(t) \leq G^{\lambda}(t)\ \forall \lambda \in \Lambda, t \in \{0,1,\ldots,T\} \quad (17)$$

Constraint (5a) now complements constraint (5) from the existing formulation, by promising to restore the world state every time a demand is made on a variable. The variable $H^{\lambda}(t)$, maintained by constraints (16) and (17), determine the desired deviation from the given profiles. The objective function has been updated to reflect that overlaps are now measured with the desired profile of usage, and there is a

T		Number of Observations								
		1	2	3	4	5	6	7	8	9
T=10	C	9	9	9	9	9	9	8.84	8.81	8.84
	U	0.39	0.417	0.394	0.399	0.406	0.35	0.36	0.484	0.429
	S	1	1	1	1	1	1	0.96	0.955	0.96
T=15	C	5.5	5.23	5.26	5.27	5.3	5.38	5.2	5.39	5.41
	U	0.007	0.008	0.007	0.008	0.006	0.002	0.008	0.01	0.009
	S	0.5	0.467	0.456	0.464	0.453	0.442	0.457	0.55	0.508
T=20	C	5.34	5.23	5.26	5.27	5.3	5.2	4.86	5.09	5.19
	U	0.004	0.004	0.004	0.004	0.003	0.001	0.004	0.008	0.006
	S	0.46	0.467	0.457	0.464	0.453	0.412	0.394	0.495	0.465
T=25	C	5.28	5.16	5.21	5.207	5.24	5.2	4.857	5.095	5.15
	U	0.003	0.003	0.003	0.003	0.002	0.001	0.003	0.007	0.004
	S	0.46	0.458	0.455	0.455	0.444	0.412	0.397	0.499	0.459

Table 1: Performance metrics w.r.t. number of observations

Time		Size of the Hypothesis Goal Set $\lvert\Psi_\alpha\rvert$					
		1	2	3	4	5	6
T=10	C	9	8.95	8.74	8.95	8.75	8.73
	U	0.345	0.364	0.54	0.228	0.55	0.42
	S	1	0.988	0.93	0.98	0.94	0.93
T=15	C	7.32	6.34	5.26	5.65	3.65	4.44
	U	0.015	0.004	0.012	0.005	0.011	0.004
	S	1	0.683	0.45	0.365	0.322	0.27
T=20	C	7.32	6.34	5.1	5.14	3.35	4.07
	U	0.009	0.002	0.007	0.003	0.006	0.002
	S	1	0.68	0.431	0.255	0.27	0.192
T=25	C	7.32	6.18	5.1	5.14	3.35	4.07
	U	0.006	0.002	0.005	0.002	0.004	0.002
	S	1	0.663	0.432	0.255	0.27	0.192

Table 2: Performance metrics w.r.t. size of the goal set

cost associated with the deviation from the real one. The revised plan now produced by the robot is shown below.

```
01 - MOVE_ROBOT_ROOM1_HALL1
02 - MOVE_ROBOT_HALL1_HALL2
03 - MOVE_REVERSE_ROBOT_HALL2_ROOM2
04 - PICK_UP_MEDKIT_ROBOT_MK1_ROOM2
05 - MOVE_ROBOT_ROOM2_HALL2
06 - MOVE_ROBOT_HALL2_HALL3
07 - CONDUCT_TRIAGE_ROBOT_HALL3
08 - MOVE_REVERSE_ROBOT_HALL3_HALL2
09 - MOVE_REVERSE_ROBOT_HALL2_ROOM2
10 - DROP_OFF_ROBOT_MK1_ROOM2
```

Notice that the robot restores the world state that the human is believed to expect, and can now communicate to him *"Can you please not use* `medkit1` *from time 7 to 9?"* based on how the real and the ideal profiles diverge, i.e. t such that $H^\lambda(t) < G^\lambda(t)$ for each resource λ.

3 Evaluation

We ran our scenario (with one human and one robot, and two medkits) on 400 problem instances, randomly generated by varying the specific (as well as the number of probable) goals of the human, and evaluated how the planner behaved with the number of observations it can start with to build its profiles. To generate the test cases, we first fix the domain description, location and goal of the agents, and the position of the resources. Then we consider 10×6 randomly generated hypothesis goal sets each of size 1 through to 6. The goals of the commander were assumed to be known to be triage related, but the location of the triage was allocated randomly, and one of the possible goals were again picked at random as the real goal. Finally for each of these problems, we generate 1-9 observations for each of these problems by simulating the commander's plan over the real goal, and plan with these observations known*a priori* the robot's plan generation process. The experiments were conducted on a Intel Xeon(R) CPU E5-1620 v2 3.70GHz$\times$8 processor with a 62.9GiB memory. The planner is available at http://bit.ly/1QHt21Q.

3.1 Scaling Up

Note that our primary contribution is the IP-formulation for planning with resource profiles, while the goal recognition component can be any off-the-shelf algorithm, and as such

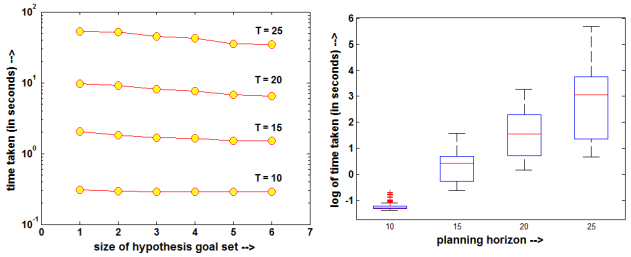


Figure 4: Performance of the planner with increasing number of possible goals and with increasing planning horizon.

we compare the scalability with respect to the planning component only. Indeed, our planner only consumes 0.2-27% (for $T = 10$ to 25 steps) of the total CPU time.

w.r.t. the Number of Agents and the Size of the Hypothesis Goal Set The IP formulation is independent of the number of agents being modeled. In fact, this is one of the major advantages of using abstractions like resource profiles in lieu of actual plans of each of the agents. On the other hand, the time spent on recognition, and on calculating the profiles, is significantly affected. However, observations on multiple agents are asynchronous, and goal recognition can operate in parallel, so that this is not a huge concern beyond the complexity of a single instance. Similarly the performance is also largely unaffected by the number of possible goals in Ψ_α, as shown in Figure 4.

w.r.t. Length of the Planning Horizon The performance of the planner with respect to the length of the planning horizon is shown in Figure 4 in terms of a box plot. This is the biggest bottleneck in the computation due to the exponential growth in the size of the IP.

3.2 Quality of the Plans Produced

Tables 1 and 2 point out some interesting aspects of planning with resource profiles. We define the U as the average conflict per step of the plan when a demand on a resource is placed by the robot, and S as the success probability per plan step that the demand is met. Notice that the average conflict goes down with increasing planning horizon T, which indicates opportunistic behavior on the part of

the robot, while the average cost C of the plans is higher for lower T, which indicates that the robot has to compromise towards higher cost plans in case of conflicts. Also note how the algorithm is quite robust with respect to the number of observations available *a priori*, indicating that the robot need not wait long to find good plans. Further, U falls drastically with higher T, which indicates that given longer plan lengths the robot is able to effectively identify lower conflict time steps to act. However, S also falls with higher T which might seems unintuitive at first, but it really means that with lesser options the robot chooses safer plans at a higher execution cost. Indeed the exact tradeoff in this behavior can be modulated by appropriately configuring the objective function of the planner.

4 Conclusions

In this paper we look at how plans may be affected by conflicts on shared resources in an environment cohabited by humans and robots, and introduce the concept of resource profiles to model the usage of such resources. We also propose a general formulation to plan in such scenarios and provide a complete framework of obtaining and using these profiles in conjunction with this planner. Finally, we show how the planner can be used to model different types of behavior of the autonomous agents. One interesting research question would be to extend the current formulation to consider nested beliefs on the agents; after all, humans are rarely completely aloof of other agents in its environment. Also, currently we only assume non-durative actions, and completely known models of the human and completely observable worlds, which we hope to relax in future works.

Acknowledgments

This research is supported in part by the ARO grant W911NF-13-1-0023, and the ONR grants N00014-13-1-0176, N00014-13-1-0519 and N00014-15-1-2027.

References

[Beaudry, Kabanza, and Michaud 2010] Beaudry, E.; Kabanza, F.; and Michaud, F. 2010. Planning with concurrency under resources and time uncertainty. In *Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, 217–222. Amsterdam, The Netherlands, The Netherlands: IOS Press.

[Blum and Furst 1995] Blum, A., and Furst, M. L. 1995. Fast planning through planning graph analysis. In *IJCAI*, 1636–1642.

[Cavallo et al. 2014] Cavallo, F.; Limosani, R.; Manzi, A.; Bonaccorsi, M.; Esposito, R.; Di Rocco, M.; Pecora, F.; Teti, G.; Saffiotti, A.; and Dario, P. 2014. Development of a socially believable multi-robot solution from town to home. *Cognitive Computation* 6(4):954–967.

[Cirillo, Karlsson, and Saffiotti 2009] Cirillo, M.; Karlsson, L.; and Saffiotti, A. 2009. Human-aware task planning for mobile robots. In *Proc of the Int Conf on Advanced Robotics (ICAR)*.

[Cirillo, Karlsson, and Saffiotti 2010] Cirillo, M.; Karlsson, L.; and Saffiotti, A. 2010. Human-aware task planning: An application to mobile robots. *ACM Trans. Intell. Syst. Technol.* 1(2):15:1–15:26.

[Helmert 2011] Helmert, M. 2011. The fast downward planning system. *CoRR* abs/1109.6051.

[Koeckemann, Pecora, and Karlsson 2014] Koeckemann, U.; Pecora, F.; and Karlsson, L. 2014. Grandpa hates robots - interaction constraints for planning in inhabited environments. In *Proc. AAAI-2010*.

[Kuderer et al. 2012] Kuderer, M.; Kretzschmar, H.; Sprunk, C.; and Burgard, W. 2012. Feature-based prediction of trajectories for socially compliant navigation. In *Proceedings of Robotics: Science and Systems*.

[Ramrez and Geffner 2010] Ramrez, M., and Geffner, H. 2010. Probabilistic plan recognition using off-the-shelf classical planners. In *In Proc. AAAI-2010*.

[Sisbot et al. 2007] Sisbot, E.; Marin-Urias, L.; Alami, R.; and Simeon, T. 2007. A human aware mobile robot motion planner. *Robotics, IEEE Transactions on* 23(5):874–883.

[Talamadupula et al. 2014] Talamadupula, K.; Briggs, G.; Chakraborti, T.; Scheutz, M.; and Kambhampati, S. 2014. Coordination in human-robot teams using mental modeling and plan recognition. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, 2957–2962.

[Tomic, Pecora, and Saffiotti 2014] Tomic, S.; Pecora, F.; and Saffiotti, A. 2014. Too cool for school adding social constraints in human aware planning. In *Proc of the International Workshop on Cognitive Robotics (CogRob)*.

[Vossen et al. 1999] Vossen, T.; Ball, M. O.; Lotem, A.; and Nau, D. S. 1999. On the use of integer programming models in ai planning. In Dean, T., ed., *IJCAI*, 304–309. Morgan Kaufmann.

Handling Advice in MDPs for Semi-Autonomous Systems

Abdel-Illah Mouaddib[1] **and Laurent Jeanpierre**[1] **and Shlomo Zilberstein**[2*†]

Abstract

This paper proposes an effective new model for decision making in situations where full autonomy is not feasible due to inability to fully model and reason about the domain. To overcome this limitation, we consider a human operator who can supervise the system and guide its operation by providing high level advice. We define a rich representation for advice and describe an effective algorithm for generating a new policy that conforms to the given advice. Advice is designed to improve the efficiency and safety of the system by imposing constraints on state visitation (either encouraging or discouraging the system to visit certain states). Coupled with the standard reward maximization criterion for MDPs, advice poses a complex multi-criteria decision problem. We present and analyze an effective algorithm for optimizing the policy in the presence of advice.

Introduction

There has been significant progress in recent years with the construction of autonomous systems for a wide range of domains from household products such as the iRobot's Roomba vacuum cleaners to space exploration vehicles. But limitations of the prevailing sensing and reasoning techniques still limit the deployment of autonomous systems in uncertain environments where a variety of unexpected events may occur. Maintaining a safe and robust behavior in such environments is a considerable challenge. In general, systems use approximate and incomplete models for planning. Even if they compute an optimal policy, the approximate nature of the model makes it hard to produce reliable operation, particularly in application domains where uncontrollable events can lead to catastrophic damage or permanent failure of the system.

A general approach to address this challenge is to develop *semi-autonomous* systems that work under the supervision of a human operator who may have more complete knowledge of the domain and better sensing abilities. The operator may intervene to correct the behavior of the system when a deviation from the desired behavior is detected, according to their knowledge. There are many examples of research efforts to support such capabilities. The multiagent systems community has long been exploring various forms of *adjustable autonomy*, allowing autonomous agents or robots to get help from humans (Côté et al. 2012; Dorais et al. 1999; Goodrich et al. 2001; Mouaddib et al. 2010). Human help could come in different forms such as *teleoperation* (Goldberg et al. 2000) or guidance in the form of *goal bias* (Côté et al. 2012). Tools to facilitate human supervision of robots have been developed. Examples include a single human operator supervising a team of robots that can operate with different levels of autonomy (Bechar and Edan 2003), or robots that operate in hazardous environments under human supervision, requiring teleoperation in difficult situations (Ishikawa and Suzuki 1997). There has also been research on mobile robots that can *proactively seek help from people* in their environment to overcome their limitations (Hüttenrauch and Severinson Eklundh 2006; Rosenthal and Veloso 2012). In robotics, researchers have started to develop robots that can autonomously identify situations in which a human operator must perform a subtask (Shiomi et al. 2008) and design suitable interaction mechanisms for the collaboration (Yanco, Drury, and Scholtz 2004).

But human intervention is often costly and should be minimized, and permanent supervision or tele-operation of a system is not desired. Therefore, it is important to develop mechanisms that allow external input to help a semi-autonomous system avoid permanent failures, situations that require external help or the activation of costly recovery mechanisms.

In this paper, we develop a formal framework allowing an operator to intervene and guide the behavior of a semi-autonomous system. Interventions consist of *advice* sent by the operator to the system to improve its behavior. Advice takes the form of a list of states to target by the system for efficiency reasons and a list of states to avoid for safety reasons. To this end, we define *advice* as sets of forbidden states, desired states and undesired states. We develop a model and algorithms to integrate such advice with an MDP-based framework and to compute on-line a new policy upon the arrival of a new advice, considering the advice as hard and soft constraints on the policy. In this paper, we focus on

[*1]University of Caen basse-Normandie, CNRS, ENSICaen, Campus II, BP5186, 14032 Caen Cedex, France {abdel-Illah.mouaddib, Laurent.Jeanpierre}@unicaen.fr

[†2]School of Computer Science, University of Massachusetts, Amherst, USA shlomo@cs.umass.edu

MDPs with terminal goal states with no loops (cyclic policies including self-loops are left for future work).

Contributions We propose a new decision-making model in which a semi-autonomous system operates while interacting with an operator in charge of extending the abilities of the system by sending it advice to improve the safely and efficiently of the mission. The advice is based on external knowledge that is not explicitly available to the system. To this end, we consider an underlying Markov decision process (MDP) where the standard reward-maximizing objective is augmented by advice from a supervisor (operator) that guides the system in avoiding risky states or visiting desired states. The paper provides:

- A formal definition of advice in terms of desired states, forbidden states and undesired states and desired actions at some states;
- A formal definition of MDP with advice, called $\mathcal{A}$-MDP;
- A formal definition of policy properties in terms of completeness, safety and satisfaction;
- A multi-criteria approach to solve an $\mathcal{A}$-MDP as a multi-criteria MDP using a lexicographic regret-based techniques;
- An efficient algorithm to derive an advice-based policy respecting the properties defined above.

Background on Markov decision process

A Markov decision process (MDP) is a mathematical tool for robust sequential decision making and planning under uncertainty. Formally, an MDP is specified by a tuple $\langle S, A, T, R, H\rangle$ as follows :

- S is a set of possible system states
- A is the set of actions
- T is the transition function: $T : S \times A \times S \rightarrow [0, 1]$ where $T(s, a, s')$ represents the probability of reaching a state s' when taking action a in state s
- R is the reward function: $R : S \rightarrow \Re$ representing the reward of the agent in state s.
- H is the horizon, representing the number of decision steps. When H is infinite, we say that the MDP is with an infinite horizon and when H is finite but unknown, we say that the MDP is with indefinite horizon.

The core decision problem for MDPs is to find a "policy" for the decision maker: a function π that specifies the action $\pi(s)$ that the decision maker should choose when in state s. The goal is to find a policy π that will maximize the expected cumulative value of the rewards, possibly discounting future rewards with a factor of γ per decision cycle. The Bellman equation defines a value function over states, $V^*(s)$, from which an optimal policy π^* can be extracted:

$$V^*(s) = argmax_{a\in A}[R(s) + \gamma \sum_{s'} T(s, a, s') \cdot V^*(s')] \quad (1)$$

We consider MDPs with indefinite horizon and terminal goal states. In this case, a process unfolds over a finite, but possibly unknown, number of steps and ends when the system reaches one of the terminal states. Many algorithms have been developed to solve such MDPs and derive an optimal policy such as value and policy iteration (Sutton and Barto 1998).

$\mathcal{A}$-MDP definition and policy properties

Definition of advice

Advice can come in the form of state visitation (telling the system to visit or not visit certain states during plan execution), such as "avoid the bridge" or "visit the park", or in the form of actions to perform or not to perform in some states, such as "don't cross the bridge" or "don't take the highway, go through the forest". To capture such guidance, we define advice $\mathcal{A}$ to be a tuple$\langle S_f, S_d, S_u, \hat{\pi}\rangle$ such that:

- $S_f \subset S$ is a set of *forbidden states* that the system must avoid;
- $S_d \subset S$ is a set of *desired* states whose visitation is preferred;
- $S_u \subset S$ is a set of *undesired* states whose avoidance is preferred;
- $\hat{\pi}$ is a *partial policy* recommending some actions in some states.

An MDP-based model with advice

An $\mathcal{A}$-MDP is a classical MDP where some states are labeled desired, undesired and forbidden and some actions are recommended at some state. More formally, an $\mathcal{A}$-MDP is a pair defined by $\langle \text{MDP}, \mathcal{A}\rangle$. A goal-based $\mathcal{A}$-MDP is defined by $\langle \text{MDP}, \mathcal{A}\rangle$ and $\mathcal{G}$, a set of terminal goal states that the agent has to reach. In this paper, we address goal-based $\mathcal{A}$-MDPs with no loops. Self-looping and cyclic policies are left for future work.

A goal-based $\mathcal{A}$-MDP presents a multi-criteria problem where we have to consider first $\hat{\pi}$, then $(\mathcal{G}, \mathcal{S}_f)$ as hard constraints to satisfy and then $(\mathcal{S}_d, \mathcal{S}_u)$, which are soft constraints to satisfy. The best policies should guarantee that a goal is reached, all forbidden states are avoided, desired states are visited and undesired states are avoided as much as possible. Hence, we face a multi-criteria MDP problem with ordered criteria.

Different techniques have been dedicated to solving such MDPs. Most of them are focused on the determination of the set of Pareto-optimal solutions. However, this set could be very large making its computation highly complex. The good news is that policies offering a good well-balanced tradeoff between criteria (Roijers, Vamplew, and Whiteson 2013) or fairly sharing the expected rewards among agents (Mouaddib, Boussard, and Bouzid 2007) present promising solution techniques to this class of MDPs. Minimizing the ordered weighted average of regrets (OWR) (Roijers, Vamplew, and Whiteson 2013) has been proposed to compute such policies. This approach is considered an extension of minmax regret technique, relaxing egalitarianism with a milder notion of fairness. The OWR approach overcomes the

limitation of minmax and weighted sum methods, which are known to reach respectively pessimistic or non-balanced solutions. OWR is then a good alternative, but it suffers from initial state dependence (its optimality depends on the initial state) and violation of the Bellman optimality principle (that each subpolicy of an optimal policy is optimal).

To this end, we propose an approach that takes advantage of the problem structure, particularly that our criteria are ordered and the initial state is known, thereby transforming OWR into a Regret Tchebychev-like measure where ideal and worst regret measures are used to guarantee the Bellman optimality principle (Mouaddib 2004; Roijers, Vamplew, and Whiteson 2013). In the following, we formally define the key concepts and policy properties using our regret value function. With such characteristics (ordered criteria, a know initial state and our regret function), the solutions we find using classical techniques such as value-iteration or linear programming are regret-optimal solutions (Roijers, Vamplew, and Whiteson 2013). We define the key ingredients of the approach below.

Probability of visitation

Definition 1 Probability of visitation *The probability of visitation of a state s when following a policy π and starting at state s_0 is:*

$$P_{vis}(s|s_0,\pi) = \sum_{t \in S} T(t,\pi(t),s) \cdot P_{vis}(t|s_0,\pi)$$

with conditions:

$$\forall \text{ terminal state } g \; P_{vis}(g|g,\pi) = 1$$

$$P_{vis}(s_0|s_0,\pi) = 1$$

Admissibility conditions of a policy

Goal states

Definition 2 *We say that a policy π is* **proper and complete** *when: $P_{vis}(\mathcal{G}|s_0,\pi) = \sum_{g \in \mathcal{G}} P_{vis}(g|s_0,\pi) = 1$*

Forbidden states

Definition 3 *We say that a policy π is* **safe** *when*

$$P_{vis}(S_f|s_0,\pi) = 0$$

$$P_{vis}(S_f|\pi) = \sum_{f \in S_f} P_{vis}(f|s_0,\pi)$$

Thus, $\forall f \in S_f, P_{vis}(f|s_0,\pi) = 0$

Admissible policy

Definition 4 *We say that a policy π is admissible when π is* **proper** *and* **complete** *and when it is* **safe**. *More formally: $\forall \pi : \pi$ is admissible when*

$$P_{vis}(\mathcal{G}|s_0,\pi) = 1 \;\; and \;\; P_{vis}(S_f|s_0,\pi) = 0$$

When admissible policies don't exist, we consider policies that maximize the probability of reaching a goal in $\mathcal{G}$ and minimize the probability of visiting S_f. To this end, we define value functions $V^{\mathcal{G},\pi}$ and $V^{S_f,\pi}$ as follows:

$$V^{\mathcal{G},\pi} = P_{vis}(\mathcal{G}|s_0,\pi)$$

$$V^{S_f,\pi} = P_{vis}(S_f|s_0,\pi)$$

We present in the following sections methods to solve these equations when an admissible policy doesn't exist.

Optimization conditions

Desired states

Definition 5 Desired states *are the states whose visitation is preferred. We say that a policy is satisfying when the expected number of visitation $\mathcal{N}_{vis}$ of S_d is maximized. The expected number of visitation $\mathcal{N}_{vis}$ of S_d when starting at state s_0 and following a policy π is as follows:*

$$\mathcal{N}_{vis}(S_d|s_0,\pi) = \sum_{s \in S_d} P_{vis}(s|s_0,\pi)$$

Undesirable states

Definition 6 Undesirable states *are the states for which avoiding visitation is preferred. We say that a policy is satisfying when the expected number of visitation $\mathcal{N}_{vis}$ of S_u is minimized. The expected number of visitation $\mathcal{N}_{vis}$ of S_u when starting at state s_0 and following a policy π is as follows:*

$$\mathcal{N}_{vis}(S_u|s_0,\pi) = \sum_{s \in S_u} P_{vis}(s|s_0,\pi)$$

Perfect satisfying condition

Definition 7 *We say that a policy π is* **perfectly satisfying** *when the following conditions hold:*
$[\forall \pi : \pi$ *is perfectly satisfying when*

$$\pi^* = argmin_\pi \mathcal{N}_{vis}(S_u|s_0,\pi) \text{ and}$$
$$\pi^* = argmax_\pi \mathcal{N}_{vis}(S_d|s_0,\pi)$$

Solving these equations can lead to many solutions or none. When there are many solution policies, we have to use the value function to select solution policies that maximize the value. However, the more interesting and more likely case is that no solution is available for the above two equations. We introduce a multi-criteria optimization technique to deal with such cases.

Partial Policy

Definition 8 Desirable actions at some states *advice recommend some actions at some states, such as "by night take the highway". Such advice is considered a partial definition of the policy π_{advice} which specifies the actions to perform at some states defined by the set S_{advice}. For such advice, the policy to follow is defined as follows:*

$$\forall s \in S_{advice}, \pi(s) = \pi_{advice}(s)$$

$$\forall s \in S - S_{advice},$$

$$\pi(s) = argmax_{a \in A}(R(s) + \sum_{s'} P(s,a,s') \cdot V(s'))$$

Overall principles and algorithms

An MDP with an advice $< \mathcal{G}, S_f, S_d, S_u, \hat{\pi} >$ can be seen as a Multi-criteria MDP where criteria have a partial order to be respected such that we should solve the MDP considering the constraints in the following order $\hat{\pi} > (\mathcal{G}, S_f) > (S_d, S_u)$. Thus, solving an MDP with advice consists of:

1. Considering $\hat{\pi}$ as a constraint on possible policies;
2. Verifying if policies exist satisfying $\mathcal{G}, S_f$ with potability 1, named admissible;
3. if admissible policies exist then satisfying S_d, S_u by maximizing $\mathcal{N}_{vis}$ of S_d and minimizing $\mathcal{N}_{vis}$ of S_u using a multi-criteria optimization based on a regret-based algorithm inspired by OWR (Roijers, Vamplew, and Whiteson 2013).
4. If admissible policies don't exist, select policies maximizing the probability to visit a goal in $\mathcal{G}$ and minimizing the probability of visiting states in S_f and then optimize S_d, S_u. To this end, we use the OWR approach where regret criteria on $\mathcal{G}$ and S_f satisfaction and on S_d, S_u satisfaction are computed and a lexicographic order over these two regret criteria is used. The use of this lexicographic method over ordered criteria guarantees a fair optimization (Roijers, Vamplew, and Whiteson 2013).

In the next sections, we present different algorithms to satisfy advice using a labeling algorithm to efficiently determine whether admissible policies exist or not and then we use a regret-based algorithm to optimally satisfy the other constraints.

Solving an MDP with $(\mathcal{G}, S_f)$ constraints when an admissible policy exists

When the advice comes in the form of (G, S_f) constraints, the solving algorithms of the MDP should derive a policy respecting these constraints. To do that, we propose the following labeling and pruning algorithm.

The algorithm we propose labels states and actions according to the satisfaction of ($\mathcal{G}$, S_f) constraints. To this end, we use the following principle: (1) all terminal goal states are labeled $+1$; (2) all forbidden states are labeled -1 ; (3) all terminal non-goal states are labeled -1. For other states, we use the following state and action labeling approach:

1. Assign a label $+1$ to goal terminal states and -1 to non-goal terminal states and forbidden states.
2. Assign a label $+1$ to an action when it leads to states labeled $+1$. Otherwise, the label of the action is -1. This is a $\min$ operator over the label of reached states with this action.
3. Assign to states the label $\max$ of the labeled action. This means that once an action ends at a goal state without visiting forbiden states this action is labeled $+1$.
4. prune all states labeled -1 and their corresponding actions.

More formally: $\forall s \in S, \forall a \in A$,

$$Label(s) = \max_{a \in A} Label(s, a)$$

input : S, A, ($\mathcal{G}$, S_f)
output : Labeled safe state and action spaces

for $s \in S_f$ **do**
  $Label(s) = -1$
end
for *terminal state s* **do**
  if $s \in \mathcal{G}$ **then**
    $Label(s) = +1$
  end
  $Label(s) = -1$
end
for $s \in S$ **do**
  for $a \in A$ **do**
    $Label(a) = \min_{s' \in S: T(s,a,s') > 0} Label(s')$
  end
  $Label(s) = \max_{a \in A} Label(a)$
end
Return Labeled A and Labeled S

Algorithm 1: The $(\mathcal{G}, S_f)$-labeling algorithm

$$\forall a \in A, \forall s \in S, Label(s, a) = \min_{s' \in S_{safe}: T(s,a,s') > 0} Label(s')$$

Thus,

$$Label(s) = \max_{a \in A} \min_{s' \in S: T(s,a,s') > 0} Label(s')$$

Theorem 1 *An admissible policy exists when the label of the initial state is* $+1$.

Proof Let s_0 be the initial state and $Label(s_0) = +1$

$$Label(s^0) = \max_{a \in A} Label(a) = +1$$

Thus $Label(a) = +1$

$$Label(a) = \min_{s^1 \in S: T(s_0,a,s_1) > 0} Label(s_1) = +1$$

Thus, $Label(s_1) = +1$ Iteratively, we can say $t \in \{0 \dots H\}$, we have $Label(s_t) = +1$ and $Label(a_1) = +1$. Note that the only states s_H with label +1 are the goal states, thus $s_H \in \mathcal{G}$. Then, when the label of initial state is +1, all the trajectories $\tau = \{s_0, a_0, s_1, a_1, \dots, s_{H-1}, a_{H-1}, s_H = goal$ have states and actions labeled +1. Then, any policy π following these trajectories will reach a state $s_H = goal$. Thus $P_{vis}(s_H = goal | s_0, \pi) = 1$. The policy π is proper and complete.

Assume that π is not safe. This means that $P_{vis}(s_f \in S_f | s_0, \pi) \neq 0$. $P_{vis}(s_f \in S_f | s_0, \pi) \neq 0$ and $P_{vis}(s_f \in S_f | s_0, \pi) = \sum_{t \in S} T(t, \pi(s_f), s_f) P_{vis}(t | s_0, \pi) \rightarrow T(t, \pi(t), s_f) > 0$. By definition, $Label(\pi(t)) = +1$, then $Label(\pi(t)) = \min_{s' \in S: T(t,\pi(t),s') > 0} Label(s') = 1 \rightarrow Label(s') = 1$, thus $Label(s_f) = 1$. This contradicts the fact that $s_f \in S_f$ and $Label(s_f)$ should be -1. Thus π is safe.

Solving an MDP with $(\mathcal{G}, S_f)$ with no admissible policy

In this section, we discuss the situation where an admissible policy doesn't exist. In this case, we use the labeling algorithm to guide the search of the best satisfying policy. To define the best satisfying policy, we define the so-called ideal

input : Labeled S, Labeled A
output : New state and action spaces
for $t \in \{0 \dots H-1\}$ **do**
  for $s^t \in S$ **do**
    for $a^t \in A$ **do**
      if $label(a^t) = -1$ **then**
        Prune a^t
        for $s^{t+1} : T(s^t, a^t, s^{t+1}) > 0$ **do**
          Prune_subtree_with_root(s^{t+1})
        end
      end
    end
  end
end
Return new state and action spaces

Algorithm 2: The $(\mathcal{G}, S_f)$-Pruning algorithm

policy which maximizes the value of states labeled $+1$ and minimizes, the value of states labeled -1.

Hence, we define $V^{\mathcal{G},*}$ and $V^{S_f,*}$ two ideal values to optimally satisfying $\mathcal{G}$ and S_f as follows :

$$V^{\mathcal{G},*}(s) = \max_{\pi} \sum_{g \in \mathcal{G}} P_{vis}(g|s,\pi) \tag{2}$$

$$V^{S_f,*}(s) = \min_{\pi} \sum_{f \in S_f} P_{vis}(f|s,\pi) \tag{3}$$

Definition 9 *We say that a policy π^* is perfect when its values are the two ideal values $V^{\mathcal{G},*}$ and $V^{S_f,*}$*

Theorem 2 *Any admissible policy is perfect.*

Proof When π is admissible, by definition of the value function, we have: $V^{\mathcal{G},\pi}(s) = 1$, which is the upper bound of $V^{\mathcal{G},*}(s)$ and $V^{S_f,\pi}(s) = 0$, which is the lower bound of $V^{S_f,*}(s)$.

To assess the value of a *non-admissible* policy π we compute its values $V^{\mathcal{G},\pi}$ and $V^{S_f,\pi}$ as follows:

$$V^{\mathcal{G},\pi}(s) = \sum_{s', label(s')=+1} T(s,\pi(s),s') \cdot V^{\mathcal{G},\pi}(s') \tag{4}$$

$$V^{S_f,\pi}(s) = \sum_{s', label(s')=-1} T(s,\pi(s),s') \cdot V^{S_f,\pi}(s') \tag{5}$$

with $\forall g \in \mathcal{G} : V^{\mathcal{G},\pi}(g) = 1$ and $\forall f \in S_f : V^{S_f,\pi}(f) = 0$

We then compare the value vectors ($V^{\mathcal{G},\pi}$, $V^{S_f,\pi}$) and ($V^{\mathcal{G},*}$, $V^{S_f,*}$). This comparison leads to a multi-criteria optimization process. To this end, we propose to use a regret Tchebychev measure to compare policies by preferring a policy over another one when it minimizes the regret. To this end, we consider ($V^{\mathcal{G},*}$, 0) respectively the best value without considering the second criterion ($V^{S_f,\pi}$) and an approximation of the worst value of $V^{\mathcal{G},\pi}$ to compute a regret measure of getting $V^{\mathcal{G},\pi}$ by policy π at a state:

$$regret(V^{\mathcal{G},\pi}) = \frac{V^{\mathcal{G},*} - V^{\mathcal{G},\pi}}{V^{\mathcal{G},*}}$$

with $regret(V^{\mathcal{G},\pi}) = 0$ when $V^{\mathcal{G},*}$ meaning there is no regret when it's impossible to accomplish the goals.

The regret of $V^{\mathcal{S}_f,\pi}$ considers ($V^{\mathcal{S}_f,*}$, 1) respectively the best value without considering the second criterion ($V^{\mathcal{G},\pi}$) and an approximation of the worst value of $V^{\mathcal{S}_f,\pi}$.

$$regret(V^{S_f,\pi}) = \frac{V^{S_f,*} - V^{S_f,\pi}}{V^{S_f,*} - 1}$$

with $regret(V^{S_f,\pi}) = 0$ when $V^{S_f,*} = 1$ meaning there is no regret when it's impossible to avoid forbidden states.

Finally, we compute the Tchebychev measure as follows:

$$regret^{\mathcal{G},S_f}(\pi) = regret(V^{\mathcal{G},\pi}) + regret(V^{S_f,\pi})$$

The best preferred policy is given by:

$$\pi^* = argmin_{\pi} regret^{\mathcal{G},S_f}(\pi)$$

Solving an $\mathcal{A}$-MDP with (S_d, S_u) soft constraints

To exploit the previous algorithms, we propose a new labeling algorithm as follows:

$$\forall s \in S_d : NLabel(s) = +1$$
$$\forall s \in S_u : NLabel(s) = -1$$

and for all states belonging to $\mathcal{G}$ and S_f are neutral and thus we label them accordingly. Thus, we propose, the following new labeling: If an admissible policy doesn't exist then

$$\forall s \in \mathcal{G} \bigcup S_f : NLabel(s) = 0$$

otherwise all states of S_f are pruned and,

$$\forall s \in \mathcal{G} : NLabel(s) = 0$$

and, for the other states we use the $minmax$ labeling algorithm as above.

From this new labeling we can compute the perfectly satisfying policy using the same principle by computing the ideal values to satisfy (S_d, S_u). Similarly to the previous section, we get:

$$V^{S_d,*}(s) = \max_{\pi} \mathcal{N}_{vis}(S_d|s_0,\pi) \tag{6}$$

$$V^{S_u,*}(s) = \min_{\pi} \mathcal{N}_{vis}(S_u|s_0,\pi) \tag{7}$$

and,

$$regret(V^{S_d,\pi}) = \frac{V^{S_d,*} - V^{S_d,\pi}}{V^{S_d,*}}$$

with $regret(V^{S_d,\pi}) = 0$ when $V^{S_d,*} = 0$

$$regret(V^{S_u,\pi}) = \frac{V^{S_u,*} - V^{S_u,\pi}}{V^{S_u,*} - 1}$$

with $regret(V^{S_u,\pi}) = 0$ when $V^{S_u,*} = 1$

$$regret^{S_d,S_u}(\pi) = regret(S_d,\pi) + regret(S_u,\pi)$$

Note that for (S_d, S_u) a simple weighted sum could be considered but to find a good balance of (S_d, S_u), we use the regret measure.

An overall solution method for $\mathcal{A}$-MDPs with advice $(\hat{\pi}, \mathcal{G}, S_f, S_d, S_u)$

Taking the constraint $\hat{\pi}$ into account, consists in redefining the value function where the value of states where the policy is defined are the expectation of applying this policy at these states. More formally,
Let $S_{\hat{\pi}}$ be the states where the policy is given and $S_{\not\hat{\pi}}$ are the other states such that $S_{\not\hat{\pi}} = S - S_{\hat{\pi}}$.
The value function for a policy π where $\hat{\pi}$ is considered is as follows:

$$V^{\pi}(s) = \begin{cases} s \in S_{\hat{\pi}} : R(s) + \sum_{s' \in S} \gamma T(s, \hat{\pi}(s), s') \cdot V^{\pi}(s') \\ s \in S_{\not\hat{\pi}} : R(s) + \sum_{s' \in S} \gamma T(s, \pi(s), s') \cdot V^{\pi}(s') \end{cases} \quad (8)$$

Then, we consider the other constraints as follows. The algorithm uses the $(\mathcal{G}, S_f)$-labeling and pruning algorithm to detect the existence of admissible policy and then considers the (S_d, S_u) soft constraints using the regret-based method to derive an admissible perfectly satisfying policy or a perfect eligible satisfying policy. This works as follows:

1. Assign to each state an action using $\hat{\pi}$ and let NS be the state spaces non-assigned.
2. Use the $(\mathcal{G}, S_f)$-labeling algorithm.
3. If the label of the initial state is $+1$, then optimize according to (S_d, S_u) in NS space by minimizing $regret^{S_d, S_u}$.
4. If the label of the initial state is -1 (no admissible policy exists), then optimize according to $(\mathcal{G}, S_f, S_d, S_u)$ in NS by using a lexicographic order on $(regret^{\mathcal{G}, S_f}, regret^{S_d, S_u})$.

More formally:

$$\pi^* = \begin{cases} \forall s \in S_{\hat{\pi}} : \hat{\pi} \\ \forall s \notin S_{\hat{\pi}} : \\ argmin_{\pi}\ regret_{\pi}^{S_d, S_u} \quad \textbf{if } label(s_0) = +1; \\ arglexmin_{\pi}(regret_{\pi}^{\mathcal{G}, S_f}, regret_{\pi}^{S_d, S_u}) \quad \textbf{otherwise} \end{cases}$$

Experimental Results

Real-robot target application

We have developed a robotic system for exploration and recognition to map and recognize an area and we developed a user-interface, Figure 1, allowing an operator to express its advice. The operator with a simple clic can show the desired, undesired or forbidden location on the map (shown in the middle of the image. This interface allows also the operator to send actions to perform. This system is dedicated to security application where robots evolve in an enemy area and receive advice on the zone to avoid (undesired), to head (desired), not to visit (forbidden) and the destination (goal).

To show how the policy computation are performed in such real examples, a grid and maze based environments have been used to assess the advice-based MDP in comparison with a classical MDP. Different situations have been developed where goals, desired, undesired and forbidden states (cells) have been expressed to evaluate and to compare the policies derived from an $\mathcal{A}$-MDP and from a classical MDP.

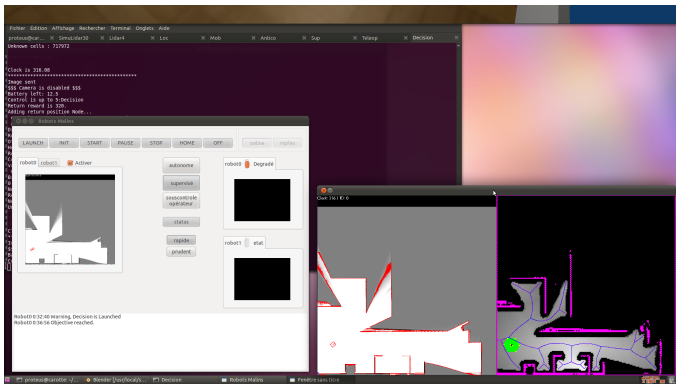


Figure 1: Interface to express advice

Model/Config	1G	1G2D2U	1G4D4U	1G8F	4G	8G
MDP	25, 9	24	25, 5	25, 7	26, 3	25, 6
$\mathcal{A}$-MDP	9, 9	9, 6	9, 5	9, 8	9, 7	9, 6

Figure 2: Table summarizing computation time (in seconds) of $\mathcal{A}$-MDP and an MDP for a 40×40 grid with 40 time units (deadline)

Performance

In this section, we evaluate the performance of $\mathcal{A}$-MDP in terms of computation time in comparison with a classical solving algorithm of MDPs to show that our approach computes policies in reasonable time that allow us to consider on-line policy computation. We also show the ability of the approach to scale up well by considering very large Maze and grid environments. We developed experiments with grids 100×100 with 100 time units leading to one million states solved in 3,3 second with our approach and 25s with a classical MDP. In general, our approach scales up well and solves different instances (as show in Table 2) in reasonable time more quickly than a classical MDP. The main reason to that is the fact that with different kind of states (goals, desired, undesired, forbidden) leads to a structured problem which is exploited with labeling and pruning algorithms to speed up the resolution. A summary of results are given in the following table with different configurations (where notation xGyDzUtF, in the table, means x goal states, y desirable states, z undesirable states and t forbidden states):

Results

We conducted a set of experiments in a grid-based environments with different situations. We present the results conducted in an environment where a goal is at the bottom right cell of the grid, two undesired cells in the bottom of the grid and along the frontier of the grid we set a set of desired cells. According to different deadlines, we can notice that when the deadlines are tight (deadline =4, Figure 3) both approaches behave in the same way by reaching the goal but violating the undesired states since the policy leads to visit them. When the deadline is larger (6 time units), Figure 4, our approach allows to reach the goal and not visit-

ing undesired states. As soon as the deadlines become larger (deadlines 8 and 10, Figures 5 and 6), $\mathcal{A}$-MDP behaves in a better way since the policy allows the robot to avoid undesired cells, reaching the goal and visiting some desired states (1 desired state for deadline 8, Figure 5 and 3 desired states for deadline 10, Figure 6). The MDP approach is an average expectation based approach, the undesired states are often visited.

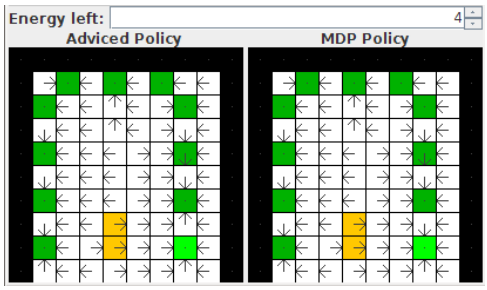


Figure 3: Derived policies from $\mathcal{A}$-MDP and classical MDP with tight deadlines in a grid (4 time units)

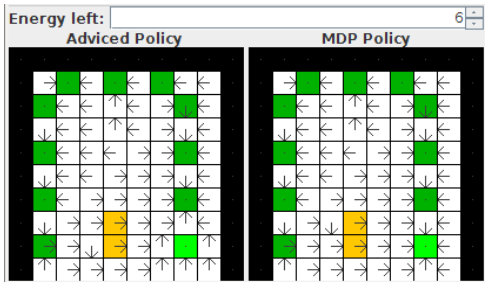


Figure 4: Derived policies from $\mathcal{A}-$MDP and classical MDP with tight deadlines in a grid (6 time units)

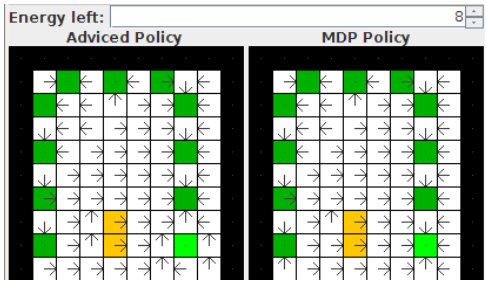


Figure 5: Derived policies from $\mathcal{A}-$MDP and classical MDP with large deadlines in a grid (deadline 8)

We conducted a set of experiments in a Maze-based environments with two situations : first situation concerns one goal state, two desired states and one undesired state and the second situation is similar the first one but we replace the undesired state with forbidden state. In the first situation $\mathcal{A}$-MDP behaves gracefully and leads to a desired policy where the undesired state is often avoided and the desired states are visited (Figure 7). In the second situation, Figure 8, the policy respects the hard constraint (forbidden state) even if this

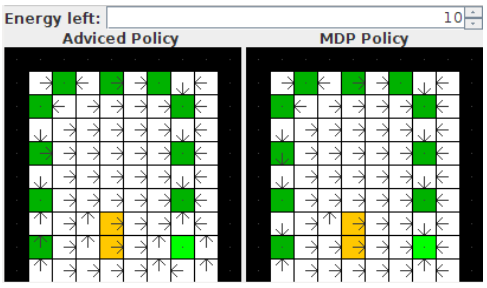


Figure 6: Derived policies from $\mathcal{A}-$MDP and classical MDP with large deadlines in a grid (deadline 10)

avoid the robot to visit the goal state while the MDP-based approach can violate this constraint.

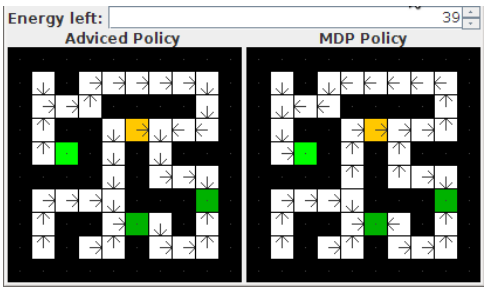


Figure 7: Derived policies from $\mathcal{A}$-MDP and classical MDP with large deadlines in a Maze with 1 goal, 2 desired states and 1 undesired state

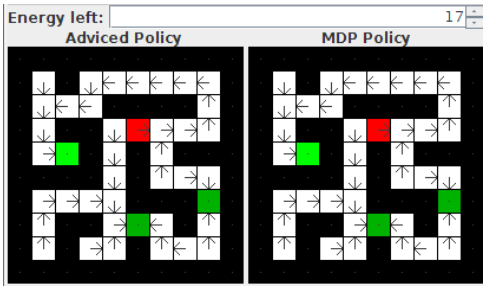


Figure 8: Derived policies from $\mathcal{A}$-MDP and classical MDP with large deadlines in a Maze with 1 goal, 2 desired states and 1 forbidden

Conclusion and future works

In this paper we present a new framework for handling advice in MDP to allow autonomous systems to consider advice from an external entity to overcome difficulties of sensing or acting. This framework presents a definition of advice in terms of goal, desired, undesired and forbidden states and how these advice could be integrated in an MDP and how to transform an MDP with advice into a multi-criteria decision-making problem. Finally, an efficient algorithm to solve such MDPs has been developed, implemented and evaluated in grid and maze environments and how it can be used in an existing robotic system. Experimental results show that $\mathcal{A}$-MDP outperforms classical MDP and that this decision

model is suitable to semi-autonomous systems and bridging the cap between them and full autonomous ones.

Acknolwledgement

This work was supported in part by NSF (USA) and ANR/DGA (France).

References

Bechar, A., and Edan, Y. 2003. Human-robot collaboration for improved target recognition of agricultural robots. *Industrial Robot: An International Journal* 30(5):432–436.

Côté, N.; Canu, A.; Bouzid, M.; and Mouaddib, A.-I. 2012. Humans-robots sliding collaboration control in complex environments with adjustable autonomy. In *Proceedings of Intelligent Agent Technology*.

Dorais, G. A.; Bonasso, R. P.; Kortenkamp, D.; Pell, B.; and Schreckenghost, D. 1999. Adjustable autonomy for human-centered autonomous systems. In *IJCAI Workshop on Adjustable Autonomy Systems*, 16–35.

Goldberg, K.; Chen, B.; Solomon, R.; Bui, S.; Farzin, B.; Heitler, J.; Poon, D.; and Smith, G. 2000. Collaborative teleoperation via the Internet. In *IEEE International Conference on Robotics and Automation*, 2019–2024.

Goodrich, M. A.; Olsen, Jr., D. R.; Crandall, J. W.; and Palmer, T. J. 2001. Experiments in adjustable autonomy. In *IEEE International Conference on Systems, Man and Cybernetics*, 1624–1629.

Hüttenrauch, H., and Severinson Eklundh, K. 2006. Beyond usability evaluation: Analysis of human-robot interaction at a major robotics competition. *Interaction Studies* 7(3):455–477.

Ishikawa, N., and Suzuki, K. 1997. Development of a human and robot collaborative system for inspecting patrol of nuclear power plants. In *Robot and Human Communication*, 118–123.

Mouaddib, A.-I.; Zilberstein, S.; Beynier, A.; and Jeanpierre, L. 2010. A decision-theoretic approach to cooperative control and adjustable autonomy. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence*, 971–972.

Mouaddib, A.-I.; Boussard, M.; and Bouzid, M. 2007. Towards a formal framework for multi-objectiove multi-agent planning. In *International Joint conference on Autonomous and MultiAgent Systems (AAMAS)*, 123–130.

Mouaddib, A.-I. 2004. Multi-criteria decision-theoretic path planning. In *IEEE International Conference on Robotics and Automation*, volume 3, 2814–2819.

Roijers, D.; Vamplew, P.; and Whiteson, S. 2013. A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research* 48:67–113.

Rosenthal, S., and Veloso, M. M. 2012. Mobile robot planning to seek help with spatially-situated tasks. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Shiomi, M.; Sakamoto, D.; Kanda, T.; Ishi, C. T.; Ishiguro, H.; and Hagita, N. 2008. A semi-autonomous communication robot: a field trial at a train station. In *Proceedings of the 3rd ACM/IEEE International Conference on Human Robot Interaction*, 303–310. New York, NY, USA: ACM.

Sutton, R. S., and Barto, A. G. 1998. *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press.

Yanco, H. A.; Drury, J. L.; and Scholtz, J. 2004. Beyond usability evaluation: Analysis of human-robot interaction at a major robotics competition. *Human–Computer Interaction* 19(1-2):117–149.