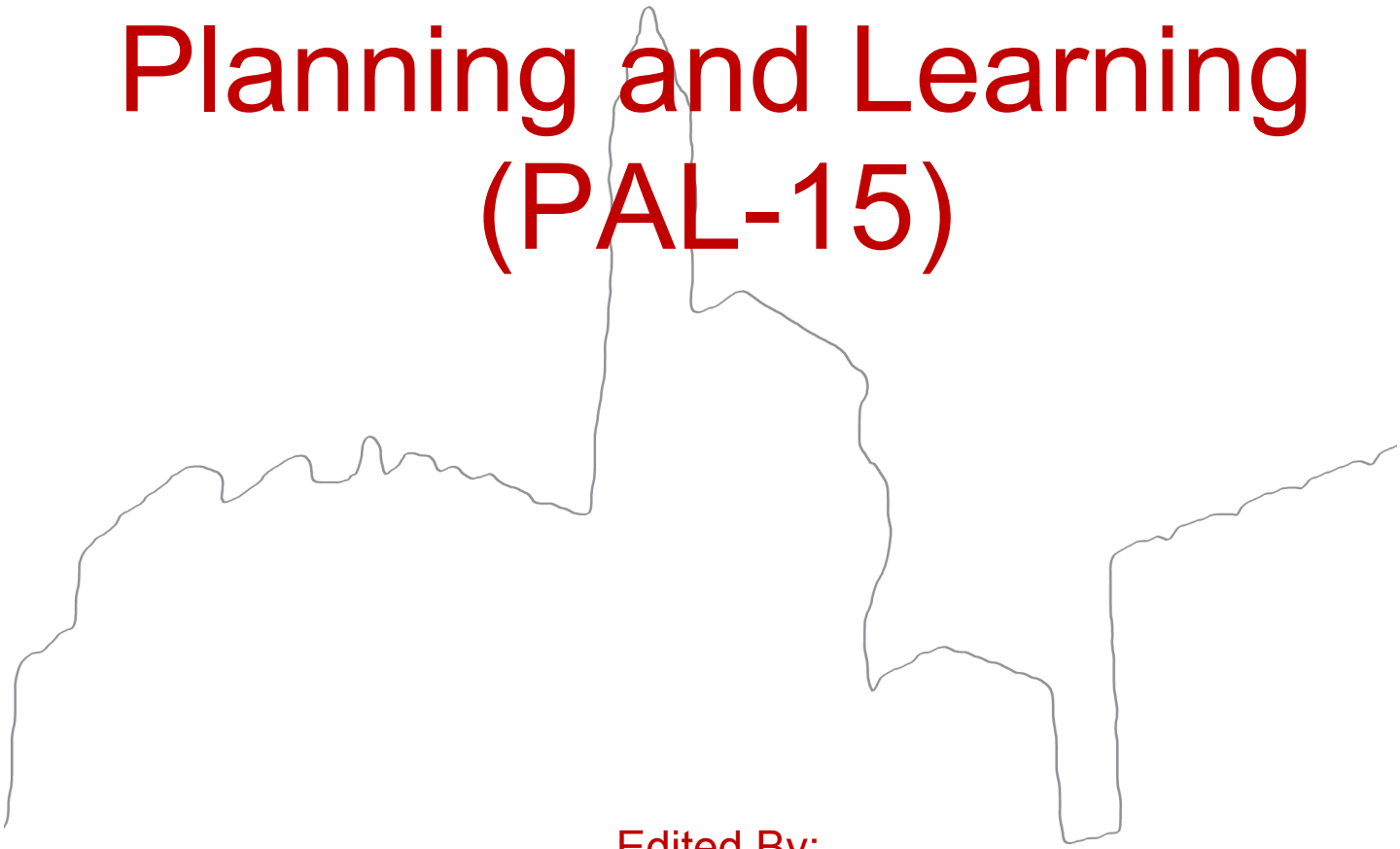
Research Workshop of the
Israel Science Foundation

Proceedings of the 5th Workshop on

Planning and Learning (PAL-15)



Edited By:

Alan Fern, Hanna Kurniawati, Scott Sanner, Nan Ye

Jerusalem, Israel 8/6/2015

Organizing Commitee

Alan Fern
Oregon State University, USA

Hanna Kurniawati
University of Queensland, Australia

Scott Sanner
Oregon State University, USA

Nan Ye
Queensland University of Technology, Australia

Program committee

Alan Fern, Oregon State University, USA
Hanna Kurniawati, University of Queensland, Australia
Scott Sanner, Oregon State University, USA
Nan Ye, Queensland University of Technology, Australia

Foreword

Learning and planning are two capabilities required for an intelligent agent and yet while they are distinct, it is often beneficial to consider them together, e.g., the interaction of learning and planning in uncertain models, learning heuristics to guide planning, using planning to solve the exploration-exploitation problem in Bayesian reinforcement learning, and adaptive Monte Carlo planning as online learning of search guidance. This workshop aims to provide a stimulating forum for researchers from both the learning community and the planning community to discuss recent advances, and potential developments on these exciting topics at the intersection of learning and planning. It continues the lineage of four previous planning and learning workshops on planning and learning in 2007, 2009, 2011, 2013. The proceedings of this workshop present some interesting papers falling into the interaction of learning and planning.

Alan, Hanna, Scott and Nan
Workshop Organizers
May 2015

Table of Contents

Value Iteration with Options and State Aggregation

Kamil Ciosek
k.ciosek@cs.ucl.ac.uk

David Silver
d.silver@cs.ucl.ac.uk

Centre for Computational Statistics and Machine Learning
University College London

Abstract

This paper presents a way of solving Markov Decision Processes that combines state abstraction and temporal abstraction. Specifically, we combine state aggregation with the options framework and demonstrate that they work well together and indeed it is only after one combines the two that the full benefit of each is realized. We introduce a hierarchical value iteration algorithm where we first coarsely solve subgoals and then use these approximate solutions to *exactly* solve the MDP. This algorithm solved several problems faster than vanilla value iteration.

Introduction

Finding solutions to discrete discounted Markov Decision Processes (MDPs) is an important problem in Reinforcement Learning. The basic problem is to obtain the optimal policy of the MDP so that the overall discounted reward obtained as we follow this policy within the MDP is maximized.[1] In this work, we do not work with the optimal policy directly but compute the optimal value function instead.

The approach we take in this paper is to modify the well-known value iteration (VI) algorithm (Bellman 1957). The basic idea of VI is to keep iterating the Bellman optimality equation. This is well-known to converge to the optimal value function. Our framework is conceptually based on a natural extension of the Bellman optimality equation where matrix models take the place of vector value functions.

In order to solve large problems, table-lookup algorithms are not practical because of the sheer number of states, which VI must loop over. Hence the need for *state abstraction*. For this work, we chose aggregation (Bertsekas 2012), which can be nicely integrated into our framework of the modified Bellman optimality equation. Algorithms based on single-step models of primitive actions are impractical, because long solution paths require many iterations of VI. Hence the need for *temporal abstraction*.[2] We solve this problem via the use of options (Sutton, Precup, and Singh 1999; Precup, Sutton, and Singh 1998) — we construct option models which can be used interchangeably with the models we have for primitive actions.

To our knowledge, this is the first paper where an algorithm using options and value iteration efficiently solves medium-sized MDPs (our 8-puzzle domain has 181441 states). Unlike prior work (Silver and Ciosek 2012), we demonstrate a modest improvement in runtime performance as well as a significant reduction in the number of iterations. Also, we have the first *convergent* VI-style algorithm where options (temporal abstraction) are combined with a framework for state abstraction, yielding far better results than the use of either idea alone. Furthermore, our algorithm is based on a principled extension of the Bellman equation. We emphasise that our algorithm converges to the *optimal value function* — although we find approximate solutions to the subgoals, these solutions are then used as inputs to solve the original MDP *exactly*, regardless of the choice of subgoals.

Background and Prior Work

State Aggregation

Consider[3] (Bertsekas 2012) an MDP with $|\mathcal{A}|$ actions; for an action a the probability transition matrix is P_a, defined by $P_a(i,j) = \gamma \text{Pr}(i_{t+1} = j | i_t = i, a_t = a)$ and the vector of expected rewards for each state is R_a, where the element corresponding to state i is defined by $R_a(i) = \text{E}\left[r_t | i_t = i, a_t = a\right]$. There are m aggregate states. We introduce the aggregation (Bertsekas 2012) matrix Φ and the disaggregation (Bertsekas 2012) matrix D of dimensions $n \times m$ and $m \times n$ respectively. Under the state aggregation approximation (Bertsekas 2012), solving the original MDP may be replaced by solving a much smaller aggregate MDP, by computing $\tilde{P}_a = DP_a\Phi$ and $\tilde{R}_a = DR_a$. The solution can then be computed by any known algorithm. VI is *convergent* because the matrices $\tilde{P}_a$ and $\tilde{R}_a$ define a valid MDP. This gives us a value function in terms of the *aggregate* states.

Options and Matrix Models

An option (Sutton 1995; Sutton, Precup, and Singh 1999; Precup, Sutton, and Singh 1998) is a tuple $\langle \mu, \beta \rangle$, consist-

[1] Our framework works for the discount factor $\gamma < 1$ as well as for those cases with $\gamma = 1$ where standard value termination converges (for example if there is a 'sink' state).

[2] Note that there is some evidence (Ribas-Fernandes et al. 2011) that subgoal-based hierarchical RL is similar to the processes actually taking place in the human brain.

[3] We refer the reader to the more elaborate introductory section in the appendix

ing of a policy μ, mapping states to actions, as well as a binary termination condition β, where $\beta(i)$ tells us whether the option terminates in state i. We will now discuss models (Sutton, Precup, and Singh 1999; Sutton 1995) for options and for primitive actions. A model consists of a transition matrix P and a vector of expected rewards R. For a primitive action a, we defined P_a and R_a in section . For options they have an analogous meaning. $R(i)$ is the expected total discounted reward given the option was executed from state i, $R(i) = E[\sum_{t=0}^{\tau} \gamma^t r_t | i_0 = i]$ where τ is the (random) duration of the option and i_0 is the starting state. The element $P(i, i')$, is the probability of the option terminating in state i', given we started in state i, considering the discounting: $P(i, i') = \sum_{\tau=1}^{\infty} \gamma^\tau \text{Pr}(\tau, i_\tau = i' | i_0 = i)$. Denote by i_0 the starting state of trajectory and by i_τ the final state. It is convenient (Sutton 1995) to arrange P and R in a block matrix of size $(n+1) \times (n+1)$, in this way: $\left[\begin{array}{c|c} 1 & 0 \\ \hline R & P \end{array}\right]$. Now model composition corresponds to matrix multiplication, i.e. if $M^{(1)}$ and $M^{(2)}$ are block matrices, $M^{(1)}M^{(2)}$ is also a block matrix corresponding to first executing the option defined in $M^{(1)}$ and then the one defined in $M^{(2)}$. In this paper, we assume that the action set $\mathcal{A} = \{A_1, \ldots, A_l\}$ is already given in this matrix format. We introduce a similar format for value functions. The value function V is represented as a vector of length $n+1$ with 1 in the first index and the values for each state in the subsequent indices. MV is a new value function, corresponding to first executing the option defined in M and then evaluating the states with V. Element $i+1$ of the vector V to state i, as does row $i+1$ of the action model. We use MATLAB notation, i.e. $V(i+1)$ is element $i+1$ of vector V and $M(i+1, :)$ is row $i+1$ of the matrix M.

Other Ways of Using Hierarchies to Improve Learning

We give a brief survey of known approaches to hierarchical learning. We stress that our approach is novel for two reasons: we *compose* macro-operators at run-time and we have *no fixed hierarchy*. This has not been done to date, except in the work on options and VI (Silver and Ciosek 2012), which introduced generalizations of the Bellman equation, versions of which we use. But it did not include state abstraction, slowing the resulting algorithm — it only produced a decrease in the *iteration count* required to solve the MDP, while we provide better *solution time*. Other approaches include using macro-operators to gain speed in planning (Korf 1985), but for deterministic systems only. Prior work on HEXQ (Hengst 2002) is largely orthogonal to ours – it focuses on hierarchy discovery, while we describe an algorithm *given* the subgoals. The work on portable options (Konidaris and Barto 2007) only discusses a flat, fixed (unlike this work) options hierarchy. MAXQ (Dietterich 1998) also involves a pre-defined controller hierarchy (the MAXQ graph)[4]. Combining the use of temporal and state abstraction was tried before, but differently from this work. The abstraction-via-statistical-testing approach (Jong and Stone 2005) only works for transfer learning — options are only constructed after the original MDP has been solved. The U-tree approach (Jonsson and Barto 2001) does not guarantee convergence to $V^\star$ for all MDPs. The modified LISP approach (Andre and Russell 2002) uses a fixed option hierarchy and the policy obtained is only optimal given the hierarchy, i.e. it may not be the optimal policy of the MDP without the hierarchy constraint.

[4]One can learn a MAXQ hierarchy (Wang, Li, and Zhou 2012), but only in a way when it is first learned and then applied.

Table-lookup Value Iteration

We begin by describing the table-lookup algorithm for computing the value function of the MDP. It is similar to the one described in previous work (Silver and Ciosek 2012), but not the same — here, termination is implemented in a different, more intuitive, way. We start with plain VI and then proceed to more complicated variants. In MATLAB notation (see section), VI can be described as follows for state i.

$$V_{(k+1)}(i+1) \leftarrow \max_a A_a(i+1, :) V_{(k)} \tag{1}$$

Here, a selects an action (control). We rewrite this update to construct a model corresponding to the optimal value function — this is not useful on its own, but will come in handy later. The following is executed for each state i.

$$\begin{aligned} a &\leftarrow \operatorname*{argmax}_a A_a(i+1, :) M_{(k)}[1, 0, \ldots, 0]^\top; \\ M_{(k+1)}(i+1, :) &\leftarrow A_a(i+1, :) M_{(k)} \end{aligned} \tag{2}$$

We note that the multiplication $M_{(k)}[1, 0, \ldots, 0]^\top$ simply extracts the total reward in the model $M_{(k)}$ (the current value function) — hence eq. is equivalent to plain VI. However, it serves an an important stepping stone to introducing *subgoals*, which is what we do next. Assume that we are, for the moment, not interested in maximizing the overall reward. Instead, we want to reach some other arbitrary configuration of states defined by the subgoal vector G of length $n+1$. The entry $i+1$ of G defines the value we associate with reaching state i. We will show later how picking such subgoals judiciously can improve the speed of the overall algorithm. Our new update, for subgoal G is the following, which we execute for each state i.

$$\begin{aligned} a &\leftarrow \operatorname*{argmax}_a A_a(i+1, :) M_{(k)} G; \\ M_{(k+1)}(i+1, :) &\leftarrow A_a(i+1, :) M_{(k)} \end{aligned} \tag{3}$$

This iteration converges (Silver and Ciosek 2012) to a model M_∞, which corresponds to the policy for reaching the subgoal G. However, this policy executes continually, it does not stop when a state with a high subgoal value of $G(i+1)$ is reached. We will now fix that by introducing the possibility of termination — in each state, we first determine if the subgoal can be considered to be reached and only then do we make the next step. This is a two-stage process, given below. First, we compute the termination condition $\beta(i)$ for each state i, according to the following equation.

$$\begin{aligned} \beta_{(k)}(i) \leftarrow \operatorname*{argmax}_{\beta_{(k)}(i) \in [0,1]} & \beta_{(k)}(i) G(i+1) + \\ & (1 - \beta_{(k)}(i)) M_{(k)}(i+1, :) G \end{aligned} \tag{4}$$

We note that this optimization is of a linear function, therefore we will either have $\beta_{(k)}(i) = 1$ (terminate in state i), or $\beta_{(k)}(i) = 0$ (do not terminate in state i). Conceptually, this update can be thought of as converting the non-binary subgoal specification G into a binary termination condition β. Once we have computed $\beta_{(k)}$, we define the diagonal matrix $\beta_{(k)} = \text{diag}(1, \beta_{(k)}(1), \beta_{(k)}(2), \ldots, \beta_{(k)}(n))$ as well as the new matrix B as follows.[5]

$$B(\beta_{(k)}, M_{(k)}) = \beta_{(k)} I + (I - \beta_{(k)}) M_{(k)}$$

Here, I is the identity matrix. B summarizes our termination condition — it behaves like model $M_{(k)}$ for the states where we do not terminate and like the identity model for the states where we do. Once we have this, we can define the actual update, which is executed for each state i.

$$\begin{aligned} a &\leftarrow \operatorname*{argmax}_a A_a(i+1,:) B(\beta_{(k)}, M_{(k)}) G; \\ M_{(k+1)}(i+1,:) &\leftarrow A_a(i+1,:) B(\beta_{(k)}, M_{(k)}) \end{aligned} \quad (5)$$

By iterating this many times, we can obtain M_∞, which will tend to go from every state to states with high values of the subgoal G. The elements of G are specified in the same units as the rewards — i.e. this algorithm will go, for the non-terminating states, to a state with a particular value of the subgoal if the value of being in the subgoal exceeds the opportunity loss of reward on the way. For the terminating states, the model will still make one step according to the induced policy (see discussion in section).

There is one more way we can speed up the algorithm — through the introduction of *initiation sets*. In this case, instead of selecting an action from the set of all possible actions, we only select an action from the set of allowed actions for a given state (the initiation set). More formally, let $S_a(i)$ be a boolean vector which has 'true' in the entries where action a is allowed is state i and 'false' otherwise. Equation then becomes the following.

$$\begin{aligned} a &\leftarrow \operatorname*{argmax}_{a:S_a(i)} A_a(i+1,:) B(\beta_{(k)}, M_{(k)}) G; \\ M_{(k+1)}(i+1,:) &\leftarrow A_a(i+1,:) B(\beta_{(k)}, M_{(k)}) \end{aligned} \quad (6)$$

The benefit of using initiation sets is that by not considering irrelevant actions, the whole algorithm becomes much faster. We defer the definition of initiation sets used to section .

Finally, we solve for several subgoals simultaneously. We use the current state of every model in every iteration, to compute the next iteration for both itself and other models. Denote our subgoals by $G^{(1)}, G^{(2)}, \ldots, G^{(g)}$ and the k-th iteration of the models trying to solve these subgoals by $M_{(k)}^{(1)}, M_{(k)}^{(2)}, \ldots, M_{(k)}^{(g)}$. Define the set $\Omega_{(k)}$ as the set of all models (macro-actions) allowed at iteration k, i.e. $\Omega_{(k)} = \{A_1, A_2, \ldots, A_l, M_{(k)}^{(1)}, M_{(k)}^{(2)}, \ldots, M_{(k)}^{(g)}\}$. This gives rise to the update given below, for each subgoal q and for each state i. We now compute the termination condition.

$$\begin{aligned} \beta_{(k)}^{(q)}(i) \leftarrow \operatorname*{argmax}_{\beta_{(k)}(i) \in [0,1]} & \beta_{(k)}(i) G^{(q)}(i+1) + \\ & (1 - \beta_{(k)}(i)) M_{(k)}^{(q)}(i+1,:) G^{(q)} \end{aligned} \quad (7)$$

The we compute one step of the algorithm according to the equation.

$$\begin{aligned} O &\leftarrow \operatorname*{argmax}_{O \in \Omega_{(k)}} O(i+1,:) B(\beta_{(k)}^{(q)}, M_{(k)}^{(q)}) G^{(q)}; \\ M_{(k+1)}^{(q)}(i+1,:) &\leftarrow O(i+1,:) B(\beta_{(k)}^{(q)}, M_{(k)}^{(q)}) \end{aligned} \quad (8)$$

Solving several subgoals simultaneously can improve the algorithm (Silver and Ciosek 2012). The immediate availability of the partial solution to every subgoal leads to faster convergence. In other words, this feature can be used to construct the macro-operator hierarchy at *run time* of the algorithm.[6] This is in contrast to many other approaches, where the hierarchy is fixed before the algorithm is run.

Combining State Aggregation and Options

We saw in section that given the aggregation[7] matrix Φ and the disaggregation matrix D, we can convert an action with the transition matrix P and expected reward vector R to an aggregate MDP by using $\tilde{P} = DP\Phi$ and $\tilde{R} = DR$. In our matrix model notation, this becomes as follows.

$$\tilde{A} = \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & D \end{array}\right] A \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & \Phi \end{array}\right], \text{ where } A = \left[\begin{array}{c|c} 1 & 0 \\ \hline R & P \end{array}\right] \quad (9)$$

This can be viewed as compressing the dynamics, given our aggregation architecture Φ of size $n \times m$, where m is the number of the aggregate states. We stress that the compressed dynamics define a valid MDP — therefore the algorithms described in the previous section are *convergent*.

The main idea of our algorithm is the following: define a subgoal, solve it (i.e. obtain a model for reaching it) and then add it to the action set of the original problem and use it as a macro-action, gaining speed. We repeat this for many subgoals. Solving subgoals is fast because we do it in the small, aggregate state space. To be precise, we pick a subgoal $\tilde{G}$ (see section for examples) and an approximation architecture Φ. We then compress our actions with eq. 9 and use compressed actions in VI according to eqs. 4 and . This gives us a model $\tilde{M}_\infty$ solving the subgoal in the aggregate state space. We want to use this model to help solve the original MDP.

However, we cannot do this directly since our model $\tilde{M}_\infty$ is defined with respect to the *aggregate* state space and has

[5] The reader will notice that our matrix B can be understood to be the expected model given the termination condition: $B(\beta_{(k)}, M_k) = E_{\beta_{(k)}}[I, M_{(k)}]$. However, in our algorithm it is enough to consider it just a matrix.

[6] By this we mean that the option models are built up in run time, possibly using other models. The subgoals are pre-defined and constant.

[7] In the work done in this paper, we used hard aggregation so that each row of Φ contains a one in one place and zeros elsewhere, and the matrix D is a renormalized version of $\Phi^\top$, so that the rows sum to one.

size $(m+1)\times(m+1)$ — we need to find a way to convert it to a model defined over the original state space, of size $(n+1)\times(n+1)$. The new model also has to be *valid*, i.e. correspond to a sequence of actions.[8]

The idea is to make the following transformation: from the aggregate model, we compute the option in the aggregate state space, we then up-scale the option to the original state space, construct a one-step model and then construct the long-term model from it. Concretely, we first compute the *option* corresponding to the model $\tilde{M}_\infty$. The option consists of the policy μ and the termination condition β. We obtain the termination condition by using eq. 4 for the aggregate states. The policy μ is obtained greedily for each aggregate state x.

$$\mu(x) = \operatorname*{argmax}_c \tilde{A}_c(x+1,:)B(\beta,\tilde{M}_\infty)\tilde{G} \qquad (10)$$

Now, we can finally build a one-step model in terms of the original state-space. We do this according to the following equation, which we use for each state i.

$$\begin{aligned} M'(i+1,:) \;=\; & (1-\beta(\phi(i)))\; A_{\mu(\phi(i))}(i+1,:) + \\ & \beta(\phi(i))\; I(i+1,:) \end{aligned} \qquad (11)$$

In the above, we denote by I the identity matrix of size $(n+1)\times(n+1)$ and by $\phi(i)$ the aggregate state corresponding[9] to the original state i. In more understandable terms, M' has rows selected by the policy μ wherever the option does not terminate and rows from the identity matrix wherever it does. Now, we do not just need a model that takes us one step towards the subgoal — we want one that takes us all the way. Therefore, we continually evaluate the option by exponentiating the model matrix, producing M'^{∞}. Now, this new model still has rows from the identity matrix where the option terminates — therefore it does not correspond to a valid combination of primitive actions. To solve this problem, we compute M'', according to the following equation (for each state i).

$$\begin{aligned} M''(i+1,:) \;=\; & (1-\beta(\phi(s)))\; M'^{\infty}(i+1,:) + \\ & \beta(\phi(s))\; A_{\mu(\phi(s))}(i+1,:) \end{aligned} \qquad (12)$$

M'' contains rows from M'^{∞} where the option does not terminate and rows dictated by the option policy where it does. This guarantees it is a valid combination of primitive actions and can be added to the action set and treated like any other action. We now run value iteration (equation 1) using the extended action set — the original actions and the subgoal models $(M'')^{(q)}$ corresponding to each subgoal q. This is s faster than using the original actions alone, even after factoring in the time used to compute the subgoal models (see section).

[8] That is why it is not possible to just upscale the model by writing: $\left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & \Phi \end{array}\right] \tilde{M}_\infty \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & D \end{array}\right]$.

[9] Note that the equation could be easily generalized to the case where the aggregation is soft — i.e. there are several aggregate states corresponding to i, simply by summing all the possibilities as weighted by the aggregation probabilities.

Observation 1. *Value Iteration with the action set* $\mathcal{A} \cup \{(M'')^{(1)},\ldots,(M'')^{(g)}\}$ *converges to the optimal value function of the MDP.*

Proof outline. The addition of subgoal macro-operators to the action set does not change the fixpoint of value iteration because the macro-operators are, by construction, compositions of the original actions. See supplement to existing work (Silver and Ciosek 2012) for a formal proof of a more general proposition.

This observation tells us that our algorithm will always exactly solve the MDP, computing $V^\star$. The worst thing that can happen is that the subgoal macro-operators will be useless i.e. the resulting value iteration will take as many iterations as without them.

Why not Use Linear Features

Looking at eqn. 9 one may ask if this is the best way to compress actions. It may seem that using *linear* features (De Farias and Van Roy 2000; Lizotte 2011; Van Roy 2005) may be better because they are more expressive and easier to come up with than Φ and D. Specifically, consider the following way of compressing actions, as an alternative to eq. 9. Define the approximation architecture $\breve{V} = \Psi w$ for modelling value functions, the sequence of which will converge to the optimal value function. We begin by defining the projection operator (Parr et al. 2008; Sorg and Singh 2010) that compresses a table-lookup model M into a model that works with linear features,

$$\breve{M}^{\Psi,\Xi}(M) = \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & \Pi^{\text{-}} \end{array}\right] M \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & \Psi \end{array}\right] \qquad (13)$$

In the above, $\Pi^{\text{-}} = (\Psi^\top\Xi\Psi)^{-1}\Psi^\top\Xi$, and Ξ is a diagonal matrix with entries corresponding to a distribution over the original states of the MDP. We introduce names for the minor matrices of the models: $\breve{M}^{\Psi,\Xi}(M) = \left[\begin{array}{c|c} 1 & 0 \\ \hline q & F \end{array}\right]$ and $M = \left[\begin{array}{c|c} 1 & 0 \\ \hline R & P \end{array}\right]$. We note that eq. 13 ensures that the model $\breve{M}^{\Psi,\Xi}(M)$ is the best approximation of the model M in the sense that it solves the optimization problems:[10] $F = \operatorname{argmin}_F \|\Psi F - P\Psi\|_\Xi$ and $q = \operatorname{argmin}_q \|\Psi q - R\|_\Xi$. In the above, the optimization is applied to the transition and reward components separately; also, each column of F is treated independently of the others. The semantics of the above is as such: each column k of F should be such as to make the entry s of the corresponding k-th column of ΨF as close as possible to the feature number k of the next state, where the index of the current state is s. Similarly, Ψq is picked so as to approximate the expected next reward for each state. In other words, F is a linear dynamical system that models the one-step dynamics on features of the Markov chain corresponding to an action. One might hope that this F and q linear dynamical system could be used in much the

[10] The norms are defined in the following way: $\|V\|_\Xi = \sqrt{V^\top\Xi V}$ and $\|A\|_\Xi = \sqrt{\operatorname{trace}(A^\top\Xi A)}$.

Figure 1: Run-times of our algorithm, plain VI and model VI. All algorithms compute $V^\star$.

Domain	plain VI	model VI	options + aggr.
Taxi (determ.)	6.43 s.	11.64 s.	4.57 s.
Taxi (stoch.)	8.30 s.	47.80 s.	4.83 s.
Hanoi (determ.)	23.45 s.	51.65 s.	11.57 s.
Hanoi (stoch.)	27.31 s.	357.52 s.	21.71 s.
8-puzzle (determ.)	100.19 s.	221.20 s.	85.94 s.

same way as the MDP compressed with state aggregation to $\tilde{P}$ and $\tilde{R}$.

But there is a problem with the compressed models defined according to eq. 13. Consider an action with the transition matrix and approximation architecture Ψ given below.

$$P = \gamma \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}; \quad \Psi = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}; \quad F = \gamma \frac{1}{3} \begin{bmatrix} 2 & 3 \\ 2 & 0 \end{bmatrix}$$

It can be easily shown that this, paired with a uniform distribution Ξ, produces the matrix F given above. But this matrix has spectrum outside of the unit circle for some $\gamma < 1$ — hence if this action is composed with itself time and again, the VI algorithm will diverge. The argument given above shows that we cannot use eq. 13 for *arbitrary* features Ψ and distributions Ξ. On the other hand, our framework based on eq. 9 does not suffer from the described divergent behaviour. Also, it does not depend on any distribution over the states, meaning there is one less parameter to the algorithm.

Experiments

We applied our approach to three domains: Taxi, Hanoi and 8-puzzle. In each case we compared several variants of VI, including our approach combining state aggregation and options. For vanilla VI we considered algorithms based on both eq. 1 (the familiar algorithm, denoted plain VI) and eq. (model VI, where complete models are constructed). Figure 1 summarises the solution times for each domain; more details are given in the following domain-specific subsections. We, however, stress beforehand that our algorithm produced a speeed-up for each of the domains we tried.

The TAXI Problem

TAXI (Dietterich 1998) is a prototypical example of a problem which combines spatial navigation with additional variables. Denote the number of states as n (here $n = 7000 + 1$) and the number of aggregate states as m (here $m = 25 + 1$). The one state is the sink state.

In our first experiment, we ran four algorithms computing the same optimal value function., one for each combination of using (or not) state aggregation and options. Consider using neither aggregation nor options — this is model VI, one iteration of which has a complexity of $O(n^2|\mathcal{A}| + n^3)$, in practice it is $O(n|\mathcal{A}|)$ because of sparsity. It takes 22 iterations to complete. Now consider the version with subgoals but no aggregation. Here, we have 5 subgoals: one for getting to each pick-up location or the fuel pump. An iteration now has complexity $O(g((|\mathcal{A}| + g)n^2 + n^3))$. Because of sparsity, this becomes $O(g((|\mathcal{A}| + g)n + n)) = O(g((|\mathcal{A}| + g)n)$. The algorithm needs 8 iterations less to converge, because subgoals allow it to make jumps. However, due to the increased cost of each iteration, the time required to converge increased. Now look at the version with aggregation (see section) and no options. There are 26 aggregate states. We map each original state to one of 25 states by taking the taxi position and ignoring other variables. Sink state (state 7001) gets mapped to the aggregate sink state (state 26). We proceed in two stages. First, all actions are compressed (eq. 9). Then, the problem is solved using model VI in this smaller state-space. This takes 330 iterations, but is fast because m is small — the complexity is $O(m^2|\tilde{\mathcal{A}}| + m^3)$. We then obtain the value function of the aggregate system and upscale it, then we use the new value function to obtain a greedy model (i.e. each row comes from the action that maximizes that row times $\bar{V}$), which we use as initialization in our iteration, which takes 3 iterations less than our original algorithm. Now consider the final version, where the benefits of aggregation and options are combined. Again, the algorithm consists of two stages. First, we use compressed actions to compute models for getting to the five subgoals. This requires 17 iterations; the complexity of each is $O(g((|\tilde{\mathcal{A}}| + g)m^2 + m^3))$, where $g = 5$. This is fast since m is small. We now upscale these models. We see that if we add the five macro-actions, we do not need the original four actions for moving, as all sensible movement is to one of the five locations. The algorithm now takes only 7 iterations to converge.[11] The run-time[12] is 6.55 s, i.e. a speed-up of 1.8 times over model VI. Results for all four versions are summarized in figure 2. We also constructed a stochastic version of the problem, with a probability of 0.05 of staying in the original state when moving. Results are qualitatively similar and are in figure 2. The speed-up from combining options with aggregation was greater at 7.1 times. We stress the main result.[13] In the deterministic case, we replace many $O(n)$ iterations with many $O(m^3)$ iterations followed by few $O(n)$ iterations. For stochastic problems, we replace many $O(n^3)$ iterations with many $O(m^3)$ iterations followed by few $O(n^3)$ iterations.

In our second experiment, as a digression from the main thrust of the paper, we tried a different approach: we can use the aggregation framework to compute an *approximate* value function, gaining speed. Our actions are compressed as defined by eq. 9, and we simply apply eq. 1. This process gives us a value function $\tilde{V}^\star$ defined over the aggregate state space (in the first case we need to extract it from the reward part of the model). We upscale this value function to the original

[11] We need an iteration to: (1) go to the fuel pump, (2) fill in fuel, (3) go to passenger, (4) pick up passenger, (5) go to destination, (6) drop off passenger. The 7th iteration comes from the termination condition.

[12] This is slightly different from the result in fig. 1 since after the models have been upscaled, we can proceed either with plain VI (as is fig. 1) or with model VI, which we do here to make the comparison fair.

[13] If the number of subgoals and actions is constant.

Figure 2: Run-times of the algorithm in the deterministic and stochastic versions of TAXI .

deter.	**no aggreg.**	**aggregation**
no options	22 iter. 11.64 s.	330 + 19 iter. 11.73 s.
options	14 iter. 78.20 s.	17 + 7 iter. 6.55 s.

stoch.	**no aggreg.**	**aggregation**
no options	30 iter. 47.80 s.	331 + 28 iter. 26.04 s.
options	18 iter. 256.04 s.	20 + 7 iter. 6.78 s.

states using the equation $\bar{V} = \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & \Phi \end{array}\right] \tilde{V}^\star$. Of course, the obtained value function $\bar{V}$ is only *approximately* optimal in the original problem. Consider a Φ with 501 aggregate states — the aggregation happens by eliminating the fuel variable and leaving others intact. The algorithm used is given by eq. , applied to compressed actions. It takes 2.94 s / 28 iterations to converge (determ.) and 3.08 s / 30 iterations (stoch.). The learned value function corresponds to a policy which ignores fuel, never visits the pump, but otherwise, if there is enough fuel, transports the passenger as intended. We have shown an important principle — if we have an aspect of a system that we feel our solution can ignore, we can eliminate it and still get an *approximate* solution. The benefit is in the speed-up. — in our case, with respect to solving the original MDP using plain VI, it is 2.2 (determ.) / 2.7 (stoch.).

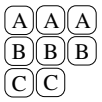
The Towers of Hanoi

For r disks, our state representation in the Towers of Hanoi is an r-tuple, where each element corresponds to a disk and takes values from $\{1, 2, 3\}$, denoting the peg.[14] There are three actions, two for moving the smallest disk and one for moving a disk between the remaining two pegs. It is known that VI for this problem takes 2^r iterations to converge. To speed up the iteration, we introduced the following state abstraction. There are $r - 2$ sub-problems of size 2,...,$r - 1$. First, we solve the problem with 2 disks, i.e. our abstraction only considers the position of the two smallest disks, ignoring the rest. There are three subgoals, one for placing the two disks on each of the pegs. Then, once we obtained three models for the subgoals, we use them to solve the subproblem of size 3, ignoring all disks except the three smallest ones. Again, there are three subgoals. We proceed until we solve the problem with r disks. For each subgoal, we need 4 iterations (Three moves and the 4th is required for the convergence criterion). The total number of iterations is $4 \times 3 \times r$, i.e. it is linear in the state space. For 8 disks this means the following speed-up: 11.57 s (with subgoals) vs. 51.65 s (model VI) vs. 23.45 s (plain VI). We note however, that the time complexity of the algorithm with subgoals is still exponential in r, because whereas the number of iterations is only linear, in each iteration we need to iterate the whole state space, which is exponential.[15] For a stochastic version, the run-times were 357.52 s for model VI, 27.31 s for plain VI and 21.71 s for computing the same optimal value function with options with aggregation.

[14]Note that the state representation itself disallows placing a larger disk on top of a smaller one.

Figure 3: The subgoal used and run-times for the 8-puzzle. All algorithms compute $V^\star$.

	iter.	**time elapsed**
model VI	32	221.20 s.
plain VI	33	100.19 s.
subgoal	25	109.51 s.
subgoal w. init. set	25	85.94 s.



The 8-puzzle

The 8-puzzle (Story 1879; **?**) is well-known in the planning community. Our subgoal is shown in figure 3.[16] 'A','B', and 'C' denote groups of tiles. The subgoal consists in arranging the tiles so that each group is in correct place (but tiles *within* each group are allowed to occupy an incorrect place). The matrix Φ is such that the original configuration of the tiles is mapped onto one where each tile is only marked with the group it belongs to. Using the subgoal alone did not result in a speed-up, so we used the notion of initiation sets (Sutton, Precup, and Singh 1999). We trained the subgoal for 9 iterations (the number 9 was obtained by trial and error), so the obtained model is only able to reach the subgoal for some starting states (the ones at most 9 steps away from the subgoal in terms of primitive actions). We upscaled the model, but this time the new model had an initiation set containing only those states from which the subgoal is reachable. The iteration we then used is plain value iteration, extended to initiation sets. The intuition behind initiation sets is that it only makes sense to use a subgoal if we are already in a part of the state space close to it. Thus, we obtained a total run-time of 85.94 seconds, which amounts to a speed-up of 1.17 over plain value iteration. The results are in figure 3.

Conclusions

We introduced new Bellman optimality equations that facilitate VI with options. These equations can be combined with state aggregation in a sound way, and therefore can be applied to the solution of medium-sized MDPs.[17] This is the first algorithm combining options and state abstraction which is *guaranteed to converge*. This is notable because other proposed approaches, notably based on linear features,

[15]However, this problem is not particular to our approach — every algorithm that purports to compute the value function *for each state* will have computational complexity at least as high as the number of such states.

[16]Other subgoals are shown in the documentation accompanying the source code. Please also consult the source code, where all subgoals are implemented.

[17]We provide software used in our experiments under GPL in the hope that others may use it for their problems.

are known to diverge even for small problems. Finally, we have shown experimentally that the benefits of options and state aggregation are only realized when they are applied *together*.

References

Andre, D., and Russell, S. J. 2002. State abstraction for programmable reinforcement learning agents. In *AAAI Conference on Artificial Intelligence / Annual Conference on Innovative Applications of Artificial Intelligence*, 119–125.

Bellman, R. 1957. *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press.

Bertsekas, D. P. 2012. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific Belmont.

De Farias, D. P., and Van Roy, B. 2000. On the existence of fixed points for approximate value iteration and temporal-difference learning. *Journal of Optimization Theory and Applications* 105:589–608.

Dietterich, T. G. 1998. The MAXQ Method for Hierarchical Reinforcement Learning. In *International Conference on Machine Learning*, 118–126.

Hengst, B. 2002. Discovering hierarchy in reinforcement learning with HEXQ . In *International Conference on Machine Learning*, volume 2, 243–250.

Jong, N. K., and Stone, P. 2005. State Abstraction Discovery from Irrelevant State Variables. . In *International Joint Conferences on Artificial Intelligence*, 752–757.

Jonsson, A., and Barto, A. G. 2001. Automated state abstraction for options using the U-tree algorithm. *Advances in neural information processing systems* 1054–1060.

Konidaris, G., and Barto, A. G. 2007. Building Portable Options: Skill Transfer in Reinforcement Learning. . In *International Joint Conferences on Artificial Intelligence*, volume 7, 895–900.

Korf, R. 1985. *Learning to Solve Problems by Searching for Macro-Operators*. Research Notes in Artificial Intelligence, Vol 5. Pitman.

Lizotte, D. J. 2011. Convergent fitted value iteration with linear function approximation. In Shawe-Taylor, J.; Zemel, R.; Bartlett, P.; Pereira, F.; and Weinberger, K., eds., *Advances in Neural Information Processing Systems 24*. 2537–2545.

Parr, R.; Li, L.; Taylor, G.; Painter-Wakefield, C.; and Littman, M. L. 2008. An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, 752–759. New York, NY, USA: ACM.

Precup, D.; Sutton, R. S.; and Singh, S. 1998. Theoretical results on reinforcement learning with temporally abstract options. In *Machine Learning: ECML-98*, volume 1398 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 382–393.

Ribas-Fernandes, J. J.; Solway, A.; Diuk, C.; McGuire, J. T.; Barto, A. G.; Niv, Y.; and Botvinick, M. M. 2011. A neural signature of hierarchical reinforcement learning. *Neuron* 71(2):370–379.

Silver, D., and Ciosek, K. 2012. Compositional planning using optimal option models. In *29th International Conference on Machine Learning*.

Sorg, J., and Singh, S. 2010. Linear options. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, 31–38. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.

Story, W. E. 1879. Notes on the "15" puzzle. *American Journal of Mathematics* 2(4):397–404.

Sutton, R.; Precup, D.; and Singh, S. 1999. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence* 112:181–211.

Sutton, R. S. 1995. TD Models: Modeling the World at a Mixture of Time Scales . In *Proceedings of the Twelveth International Conference on Machine Learning*, 531–539. Morgan Kaufmann.

Van Roy, B. 2005. TD(0) Leads to Better Policies than Approximate Value Iteration . In Weiss, Y.; Schölkopf, B.; and Platt, J., eds., *Advances in Neural Information Processing Systems 18*. 1377–1384.

Wang, H.; Li, W.; and Zhou, X. 2012. Automatic discovery and transfer of maxq hierarchies in a complex system. In *ICTAI*, 1157–1162.

Appendix

In this appendix, we discuss background information concerning state aggregation for MDPs, adapted to the notation of our paper. This is necessary because Bertsekas' original notation is difficult to apply to our work. We stress that the ideas presented in this appendix are entirely due to Bertsekas (Bertsekas 2012).

We are concerned with an MDP which has $|\mathcal{A}|$ actions, and for an action a the probability transition matrix is P_a, defined by $P_a(i,j) = \gamma \Pr(i_{t+1} = j | i_t = i, a_t = a)$ and the vector of expected rewards for each state is R_a, defined by $R_a(i) = \mathrm{E}\left[r_t | i_t = i, a_t = a\right]$. There are m aggregate states. In addition, we introduce two matrices[18] defining the approximation architecture: the *aggregation matrix* Φ and the *disaggregation matrix* D. The matrix Φ has dimensions $n \times m$ and the matrix D has dimension $m \times n$. It is useful to think about these matrices as conversion operators: the matrix Φ converts a value function defined over the aggregate states into one defined over the original states; conversely, the matrix D converts a value function defined over the original states into one defined over the aggregate states. There are no conditions on these matrices other than the rows have to sum to one, as they are probability distributions modeling, for Φ, the degree by which each state is represented by various aggregate states and, for D, the degree to which a certain aggregate state corresponds to various original states. Having defined the matrices, we can define our first approximation step. The Bellman optimality operator in the original MDP is called T, and is defined by $(TV)(i) = \max_a (P_a V)(i) + R_a(i)$ and the optimum value function $V^\star$ satisfies the fixpoint equation $V^\star = TV^\star$. Now, the approximation consists in solving the following equation instead (we will see later that this is not solved exactly and further approximation is necessary).

$$\tilde{V}^\star = DT(\Phi \tilde{V}^\star) \tag{14}$$

[18] We employ the names introduced by Bertsekas (Bertsekas 2012).

In the above, we use $\tilde{\cdot}$ to denote the aggregate problem. We note that this equation operates on a shorter value function — $\tilde{V}^\star$ has entries corresponding to *aggregate* states. The idea is, of course that the number of aggregate states is tractable, so we can compute $\tilde{V}^\star$. However, we need to reformulate the equation since in its present form it contains the operator T, which still operates on the original states. To do so, we expand the definition of T, to obtain the following state-wise equation, for the aggregate state x.

$$\tilde{V}^\star(x) = \sum_i d_{xi} \left(\max_a P_a(i,:)\Phi\tilde{V}^\star + R_a(i) \right)$$

This equation leads to the following iterative algorithm, which computes $\tilde{V}^\star$ as $k \to \infty$.

$$\tilde{V}_{(k+1)}(x) = \sum_i d_{xi} \left(\max_a P_a(i,:)\Phi\tilde{V}_{(k)} + R_a(i) \right)$$

In the above, $P_a(i,:)$ denotes the row number i of the probability transition matrix corresponding to action a (in terms of the original states). Value functions are assumed to be column vectors. In order to be able to operate exclusively with objects that have dimensionality corresponding to the number of *aggregate* states, we introduce another approximation and namely we do the following.

$$\tilde{V}_{(k+1)}(x) = \max_a \sum_i d_{xi} \left(P_a(i,:)\Phi\tilde{V}_{(k)} + R_a(i) \right)$$

We note that this approximation is exact if states mapping to a single aggregate state all have the same optimal action. Now, we can reformulate the equation in the following way.

$$\begin{aligned} \tilde{V}_{(k+1)}(x) &= \max_a D(x,:)P_a\Phi\tilde{V}_{(k)} + D(x,:)R_a \\ &= \max_a (\tilde{P}_a\tilde{V}_{(k)})(x) + \tilde{R}_a(x) \end{aligned} \tag{15}$$

In the above, $D(x,:)$ denotes the row of D corresponding to aggregate state x and P_a is the probability transition matrix corresponding to action a in the original MDP. Now, we note that solving the above equation is equivalent to solving a modified MDP with actions corresponding to the original actions, probability transition matrices given by $\tilde{P}_a = DP_a\Phi$ and expected reward vectors given by $\tilde{R}_a = DR_a$. The states of the modified MDP are the aggregate states.

Therefore, under our two explained approximations, solving the original MDP may be replaced by solving a much smaller aggregate MDP, by computing $\tilde{P}_a$ and $\tilde{R}_a$. The solution can then be computed by any known algorithm, although in this paper we focus only on VI. We emphasize that the VI is *convergent* because the matrices $\tilde{P}_a$ and $\tilde{R}_a$ define a valid MDP. We stress again that this involves two approximations: first, we are solving a modified Bellman equation 14 that utilizes state aggregation and second, we move the $\max$ operator outside of the sum in equation 15.

Path Finding under Uncertainty through Probabilistic Inference

David Tolpin, Brooks Paige, Jan Willem van de Meent, Frank Wood
University of Oxford
{dtolpin,brooks,jwvdm,fwood}@robots.ox.ac.uk

Abstract

We introduce a new approach to solving path-finding problems under uncertainty by representing them as probabilistic models and applying domain-independent inference algorithms to the models. This approach separates problem representation from the inference algorithm and provides a framework for efficient learning of path-finding policies. We evaluate the new approach on the Canadian Traveller Problem, which we formulate as a probabilistic model, and show how probabilistic inference allows high performance stochastic policies to be obtained for this problem.

Introduction

In planning under uncertainty the objective is to find the optimal policy — a policy that maximizes the expected reward. In most interesting cases the optimal policy cannot be found exactly, and approximation schemes are used to discover the policy, either represented explicitly or as an implicit property of the planning algorithm, through *reinforcement learning*. Approximation schemes include value/policy iteration, Q-learning, policy gradient methods (Sutton and Barto 1998), as well as methods based on heuristic search (Bonet and Geffner 2001) and Monte Carlo sampling such as MCTS (Kocsis and Szepesvári 2006; Browne et al. 2012).

Domain-independent planning algorithms (Bonet and Geffner 2001; Haslum, Bonet, and Geffner 2005; Helmert 2006) can be applied to different domains with little modification, however for many applications domain-dependant techniques are still critical in order to obtain a high performance policy, and put the burden of implementation on the domain expert formulating the planning problem.

The framework of probabilistic inference (Pearl 1988) proposes solutions to a wide range of Artificial Intelligence problems by representing them as *probabilistic models*. Efficient domain-independent algorithms are available for several classes of representations, in particular for graphical models (Lauritzen 1996), where inference can be performed either exactly and approximately. However, graphical models typically require that the full graph of the model to be represented explicitly, and are not powerful enough for problems where the state space is exponential in the problem size, such as the generative models common in planning (Szörényi, Kedenburg, and Munos 2014).

Probabilistic programs (Goodman et al. 2008; Mansinghka, Selsam, and Perov 2014; Wood, van de Meent, and Mansinghka 2014) can represent arbitrary probabilistic models, efficient approximate inference algorithms have recently emerged (Wingate, Stuhlmüller, and Goodman 2011; Wood, van de Meent, and Mansinghka 2014; Paige et al. 2014). In addition to expressive power, probabilistic programming separates modeling and inference, allowing the problem to be specified in a simple language which does not assume any particular inference technique.

In this paper, we show a connection between probabilistic inference and path finding, which allows many path-finding problems to be cast as inference problems using probabilistic programs. Based on this connection, we provide a generic scheme for expressing a path-finding problem as a probabilistic program that infers the path-finding policy. We illustrate this generic scheme by its application to the Canadian Traveller Problem (Papadimitriou and Yannakakis 1991; Bar-Noy and Schieber 1991; Nikolova and Karger 2008). In the empirical evaluation, we show that high performance stochastic policies can be obtained using domain-independent inference techniques. In the concluding section, we discuss other possible areas of application of probabilistic programming in planning, as well as possible difficulties.

Preliminaries

Probabilistic Programming

Probabilistic programs are regular programs extended by two constructs (Gordon et al. 2014):

- The ability to draw random values from probability distributions.
- The ability to condition values computed in the programs on probability distributions.

A probabilistic program implicitly defines a probability distribution over the program's output. Formally, we define a probabilistic program as a stateful deterministic computation $\mathcal{P}$ with the following properties:

- Initially, $\mathcal{P}$ expects no arguments.

- On every call, $\mathcal{P}$ returns either a distribution F, a distribution and a value (G, y), a value z, or $\perp$.
- Upon returning F, $\mathcal{P}$ expects a value x drawn from F as the argument to continue.
- Upon returning (G, y) or z, $\mathcal{P}$ is invoked again without arguments.
- Upon returning $\perp$, $\mathcal{P}$ terminates.

A program is run by calling $\mathcal{P}$ repeatedly until termination. Every run of the program implicitly produces a sequence of pairs (F_i, x_i) of distributions and drawn from them values of latent random variables. We call this sequence a *trace* and denote it by $\boldsymbol{x}$. A trace induces a sequence of pairs (G_j, y_j) of distributions and values of observed random variables. We call this sequence an *image* and denote it by $\boldsymbol{y}$. We call a sequence of values z_k an *output* of the program and denote it by $\boldsymbol{z}$. Program output is deterministic given the trace.

By definition, the probability of a trace is proportional to the product of the probability of all random choices $\boldsymbol{x}$ and the likelihood of all observations $\boldsymbol{y}$:

$$p_{\mathcal{P}}(\boldsymbol{x}|\boldsymbol{y}) \propto \prod_{i=1}^{|\boldsymbol{x}|} p_{F_i}(x_i) \prod_{j=1}^{|\boldsymbol{y}|} p_{G_j}(y_j) \tag{1}$$

The objective of inference in probabilistic program $\mathcal{P}$ is to discover the distribution of $\boldsymbol{z}$.

Several implementations of general probabilistic programming languages are available (Goodman et al. 2008; Mansinghka, Selsam, and Perov 2014; Wood, van de Meent, and Mansinghka 2014). Inference is usually performed using Monte Carlo sampling algorithms for probabilistic programs (Wingate, Stuhlmüller, and Goodman 2011; Wood, van de Meent, and Mansinghka 2014; Paige et al. 2014). While some algorithms are better suited for certain inference types, most can be used with any valid probabilistic program.

Canadian Traveller Problem

Canadian Traveller Problem (CTP) was introduced in (Papadimitriou and Yannakakis 1991) as a problem of finding the shortest travel distance in a graph where some edges may be blocked. There are several variants of CTP (Bar-Noy and Schieber 1991; Nikolova and Karger 2008; Bnaya, Felner, and Shimony 2009); here we consider the *stochastic* version. In the stochastic CTP we are given

- Undirected weighted graph $G = (V, E)$.
- The initial and the final location nodes s and t.
- Edge weights $w : E \to \mathcal{R}$.
- Traversability probabilities: $p_o : E \to (0, 1]$.

The actual state of each edge is fixed for every problem instance but becomes known only upon reaching one of the edge vertices. The goal is to find a *policy* that minimizes the expected *travel distance* from s to t. The travel distance is the sum of weights of all traversed edges during the travel, where traversing in each direction is counted separately.

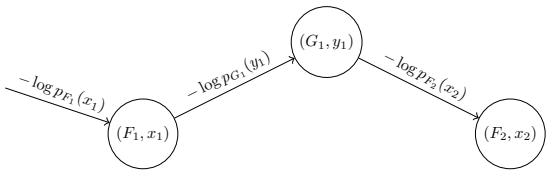


Figure 1: A path in the graph of a probabilistic program

CTP problem is PSPACE-hard (Fried et al. 2013), however a number of heuristic algorithms were proposed, including high-quality policies based on Monte Carlo methods (Eyerich, Keller, and Helmert 2010). Policies are empirically compared by averaging the distance travel over multiple instantiations of the actual states of the edges (open or blocked) according to the traversal probabilities. Since the travel distance is defined only for instance where a path between s and t exists, instantiations in which t cannot be reached from s are ignored.

A trivial travel policy is realized by traversing the problem graph in a depth-first order until the final location is reached. The expected travel distance of the policy is bounded from above by the sum of weights of all edges in the graph by noticing that every edge is traversed at most once in each direction, and at most half of the edges are traversed on average.

Duality between Path Finding and Probabilistic Inference

We shall now show a connection between path finding and probabilistic inference. This connection was noticed earlier (Shimony and Charniak 1991) and was used to search for the *maximum a-posteriori* probability (MAP) assignment in graphical models using a best-first search algorithm. Here we further extend the analogy and establish a bilateral correspondence between inferring the distribution defined by a probabilistic model and learning the optimal policy in a path-finding problem.

We proceed in two steps. First, following earlier work, we establish a connection between a MAP assignment and the shortest path. Then, based on this analogy, we explain how discovering the optimal policy in a generative model can be translated into inferring the output distribution of a probabilistic program and vice versa.

Inference on probabilistic programs computes a representation of distribution (1). An equivalent form of (1) is obtained by taking logarithm of both sides:

$$\log p_{\mathcal{P}}(\boldsymbol{x}) = \sum_{i=1}^{|\boldsymbol{x}|} \log p_{F_i}(x_i) + \sum_{j=1}^{|\boldsymbol{y}|} \log p_{G_j}(y_j) + C \tag{2}$$

where C is a constant that does not depend on $\boldsymbol{x}$. To find the MAP assignment $\boldsymbol{x}_{MAP}$, one must maximize $\log p(\boldsymbol{x})$. One can view $\boldsymbol{x}$ as a specification of a path in a graph where each node corresponds to either (F_i, x_i) or (G_j, y_j), and the costs of edges entering (F_i, x_i) or (G_j, y_j) is $-\log p_{F_i}(x_i)$ o $-\log p_{H_j}(y_j)$, correspondingly (Figure 1). Then, finding

the MAP assignment is tantamount to finding the trace $\boldsymbol{x}$ that produces the shortest path. (Shimony and Charniak 1991; Sun, Druzdzel, and Yuan 2007) use this correspondence in their MAP algorithms for graphical models.

We shall turn now to a more general case when the MAP assignment of a part of the trace $\boldsymbol{x}^\theta$ is inferred. In a probabilistic program, this is expressed by selecting $\boldsymbol{x}^\theta$ as the program output, $\boldsymbol{z} \leftarrow \boldsymbol{x}^\theta$. The distribution of $\boldsymbol{z}$ is marginalized over the rest of the trace $\boldsymbol{x}^{\neg\theta} = \boldsymbol{x} \setminus \boldsymbol{x}^\theta$, and finding the MAP assignment for $\boldsymbol{x}^\theta$ corresponds to finding the mode of the output distribution:

$$\begin{aligned} \boldsymbol{x}^\theta_{MAP} &= \arg\max p_{\mathcal{P}}(\boldsymbol{z}) \\ &\propto \arg\max_{\boldsymbol{x}^{\neg\theta}} \left[\prod_{i=1}^{|\boldsymbol{x}^\theta|} p_{F_i^\theta \boldsymbol{x}_i^\theta} \prod_{j=1}^{|\boldsymbol{y}|} p_{G_j}(y_j) \right] p_{\mathcal{P}}(\boldsymbol{x}^{\neg\theta}) d\boldsymbol{x}^{\neg\theta}, \end{aligned} \tag{3}$$

where the integrand in equation (3) depends on $\boldsymbol{x}^{\neg\theta}$. Just like in the case of MAP assignment to all random variables, equation (3) corresponds to a path finding problem: $\boldsymbol{x}^\theta$ can be viewed as a policy, and determining $\boldsymbol{x}^\theta_{MAP}$ corresponds to learning a policy which minimizes the expected path length

$$\mathbb{E}_{\boldsymbol{x}^{\neg\theta}} \left[-\sum_{i=1}^{|\boldsymbol{x}^\theta|} \log p_{F_i^\theta}(x_i^\theta) - \sum_{j=1}^{|\boldsymbol{y}|} \log p_{G_j}(y_j) \right] \tag{4}$$

While in principle policy learning algorithms could be used for MAP estimation, a greater potential lies, in our opinion, in casting planning problems as probabilistic programs and learning the optimal policies by estimating the modes of the programs' distributions. We suggest to adopt the Bayesian approach, according to which prior beliefs are imposed on policy parameters, and the optimal policy is learned through inferring posterior beliefs by conditioning the beliefs on observations. We explore this approach in the next section.

Stochastic Policy Learning through Probabilistic Inference

We have shown that in order to infer the optimum policy, a probabilistic program for policy learning should run the agent on the distribution of problem instance and policies, and compute probability of each execution such that the logarithm of the probability is equal to the negated travel cost. The generic program shown in Algorithm 1 achieves this by randomly drawing problem instances and policies from their distributions supplied as program arguments (lines 1 and 2) and updating the log probability of the sample (line 4) by calling OBSERVE. OBSERVE adds the log probability of its first argument, the value, with respect to its second argument, the distribution. Consequently, the log probability of the output policy

Algorithm 1 Policy learning through probabilistic inference.

Require: $agent$, $Instances$, $Policies$
1: $instance \leftarrow$ DRAW($Instances$)
2: $policy \leftarrow$ DRAW($Policies$)
3: $cost \leftarrow$ RUN($agent$, $instance$, $policy$)
4: OBSERVE(1, Bernoulli(e^{-cost}))
5: PRINT($policy$)

$$\begin{aligned} &\log p_{\mathcal{P}}(policy) = \\ &\quad \log p_{Policies}(policy) + \log e^{-cost(policy)} + C \\ &= \underline{-cost(policy)} + \log p_{Policies}(policy) + C \end{aligned} \tag{5}$$

When policies are drawn from their distribution uniformly, $\log p_{Policies}(policy)$ is the same for any policy, and does not affect the distribution of policies specified by the probabilistic program:

$$\log p_{\mathcal{P}}(policy) = -cost(policy) + C' \tag{6}$$

In practice, this is achieved by using a uniform distribution on policy parameters, such as the uniform continuous or discrete distribution for scalars, the categorical distribution with equal choice probabilities for discrete choices, or the symmetric Dirichlet distribution with parameter 1 for real vectors. Alternatively, if different policies have different probabilities with respect to the distribution $Policies$ from which the policies are drawn, their log probabilities (taken with the opposite sign) have the interpretation of the costs of the corresponding policies and provide a means for specifying preferences of the model designer with respect to different policies. In either case, the optimal policy is approximated by estimating the mode of the program output.

When policies are drawn uniformly, the scale of the travel cost does not affect the choice of optimal policy. However, as follows from equation (6), the shape of the probability density (or probability mass for discrete distributions) depends on the cost scale — the higher the cost, the sharper the shape. Thus, by altering the cost scale we can affect the performance of the inference algorithm: on one hand, the mode estimate of a sharper function can be computed with higher accuracy, on the other hand, when $p_{\mathcal{P}}(policy)$ changes too fast with its argument in the high probability region, approximate inference algorithms converge slowly. The right scale depends on the probabilistic program, and finding the most appropriate scale is a parameter optimization problem.

Note that the probabilistic program for policy learning is independent of the inference algorithm which would be used to obtain the results. The programmer does not need to make any assumptions about the way the mode of the output distribution is estimated, and even whether approximate or exact inference (if appropriate) is performed.

Case Study: Canadian Traveller Problem

We evaluated the proposed policy learning scheme on the Canadian Traveller Problem (Algorithm 2). The algorithm draws CTP problem instances from a given graph with traversability of each edge randomly selected according to the probabilities p, and learns a stochastic policy based on depth-first search — the policy is specified by a vector of probabilities of selecting each of the adjacent edges in every

node. When the policy is realized, the selection probabilities are conditioned such that only open unexplored edges are selected, in accordance with the base depth-first search traversal. $Dirichlet(\mathbf{1}^{\deg(v)})$ is a uniform distribution, hence the

Algorithm 2 Learning stochastic policy for the Canadian traveller problem

Require: CTP(G, s, t, w, p)
1: $instance \leftarrow$ DRAW(CTP(G, w, p))
2: **for** $v \in V$ **do**
3: $\quad policy(v) \leftarrow$ DRAW(Dirichlet$(\mathbf{1}^{\deg(v)})$)
4: **end for**
5: **repeat**
6: $\quad (reached, distance) \leftarrow$ STDFS$(instance, policy)$
7: **until** $reached$
8: OBSERVE(1, Bernoulli$\left(e^{-distance}\right)$)
9: PRINT$(policy)$

log probability of a trace is equal to the path cost taken with the opposite sign. STDFS (line 6) is a flavour of depth-first search which enumerates node children in a random order according to the policy for the current node. An optimal policy is expected to assign a higher probability to edges leading to shorter paths having lower probability to become blocked.

To assess the quality of learned policies we generated several CTP problem specifications by triangulating a randomly drawn set of either 50 or 20 nodes from Poisson-distributed points on a unit square. The average DFS travel cost in fully traversable instances was 7.9 for 50 node instances, and 5.7 for 20 node instances. The same traversal probability in the range [0.35, 1.0] is assigned to every edge of the graph (the bond percolation threshold for Delaunay triangulation is ≈0.33 (Becker and Ziff 2009), hence instances with $p < 0.3$ are disconnected with high probability). A 50 node instance is shown in Figure 2. The s and t nodes are marked by the red circles, and edge weights are equal to the Euclidean distances between the nodes.

Lightweight Metropolis-Hastings (Wingate, Stuhlmüller, and Goodman 2011) was used for inference. We learned a policy for each problem specification by running the inference algorithm for 10,000 iterations. Then, we evaluated policies returned at different numbers of iterations on 1,000 randomly drawn instances to estimate the average travel cost. The average computation time of learning and evaluation per instance was ≈80s on Intel Core i5 CPU.

The results are shown in Figure 3, where the solid lines correspond to the average travel cost over the set of problems of the corresponding size, and dashed lines to 95% confidence intervals. For both 50 and 20 node problems, the policy mostly converged after ≈1000 iterations, achieving 50–80% improvement compared to the uniform stochastic policy. While a further refinement of the policy is possible, a different type of policy should be learned to obtain significantly better results, for example, a deterministic policy which takes online information into account. This, however, would complicate the probabilistic program which we chose to keep as simple as possible — the actual implementation

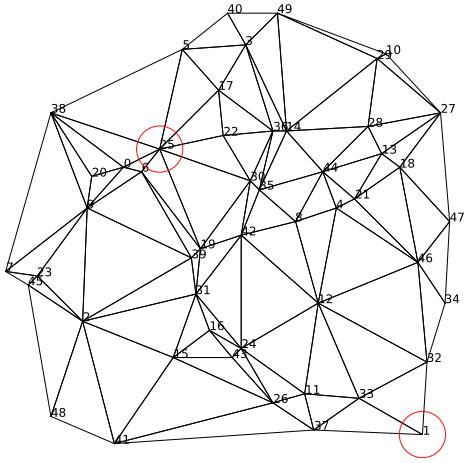


Figure 2: An instance of CTP with 50 nodes. Initial (1) and final (25) locations are marked by red circles; edge weights are Euclidean distances between edge vertices.

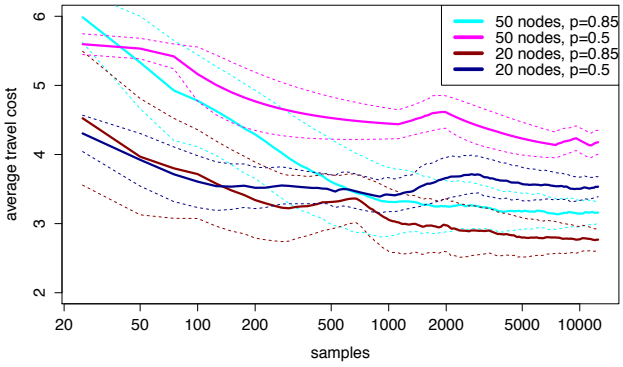


Figure 3: Average travel cost vs. number of samples for problems with 50 and 20 nodes and traversability probabilities 0.85 and 0.5. The policies mostly converged after ≈1000 samples.

of the program is just above 100 lines of code, including the implementation of DFS.

A learned policy for a 50 node problem is visualized in Figure 4. Edge widths correspond to the confidence about the policy for the edge. Edges with higher precision (lower variance) of the policy are broader. Edge color is blue when a traversal through the edge is much more probable in one than in the other direction, and green when traversal in either direction has the same probability, with shades of green and blue reflecting how directed the edge is. As we would expect in a converged policy, edges in the center of the graph are thicker, that is, more explored, than at the periphery, where changes in the policy are less likely to affect the average

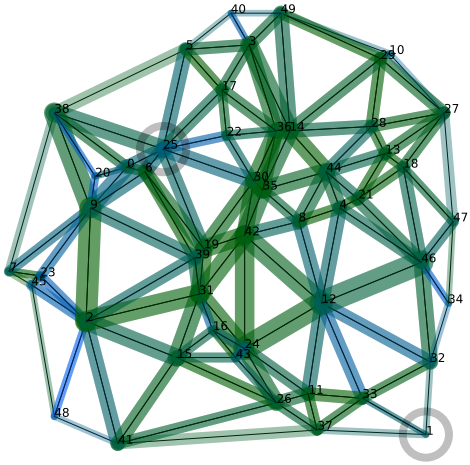


Figure 4: Visualization of policy learned for blocking probability 0.5 on instance in Figure 2. Broader edges correspond to more explored components of the policy.

travel cost. Bright blue (uni-directional) edges are mostly radial relative to the direction from the initial position (node 1) to the goal (node 25), and many well-explored tangential edges are green (bi-directional). This corresponds to an intuition about the policy — traversals through radial edges are mostly in the direction of the goal, and through the tangential edges in either direction to find an alternative route when the edge leading to the goal is blocked.

Discussion

We introduced a new approach to policy learning based on casting a policy learning task as a probabilistic program. The main contributions of the paper are:

- Discovery of bilateral correspondence between probabilistic inference and policy learning for path finding.
- A new approach to policy learning based on the established correspondence.
- A realization of the approach for the Canadian traveller problem, where improved policies were consistently learned by probabilistic program inference.

The proposed approach can be extended to many different planning problems, both in well-known path-finding applications and in other domains involving policy learning under uncertainty; Partially observable Markov Decision Processes and generalized Multi-armed bandit settings are just two examples. At the same time, the exposure of probabilistic programming tools to different domains and new applications is challenging. These tools were initially developed with certain applications in mind. Our limited experience shows that the probabilistic programming paradigm scales well to new applications and larger problems. However, as more problems are approached using the probabilistic programming methodology, apparent weaknesses and limitations are uncovered, and a more powerful and flexible inference algorithm will have to be developed.

The policy learning algorithm presented here follows the offline learning scheme — the policy is selected before acting, and then used unmodified until the goal is reached. Although this is, indeed, the easiest way to cast policy learning as probabilistic inference, online learning can also be implemented so that when additional computation during acting is justified by the time cost, the policy is updated based on the information gathered online, as in some of state-of-the-art algorithms for CTP (Eyerich, Keller, and Helmert 2010). Moreover, the time cost of updating the policy incrementally based on the new evidence is lower than of inferring a new policy due to the any-time nature of Bayesian updating. Online inference is a subject of ongoing research in probabilistic programming.

By performing inference on a probabilistic program, we obtain a representation of *distribution of policies* rather than a single policy. We then use this distribution to select a policy. When the inference is performed approximately, which is a common case, the expected quality of the selected policy improves with more computation. In the most basic setting, a fixed threshold on the number of iterations of the inference algorithm can be imposed. In general, however, determining when to stop the inference and commit to a particular policy, whether in offline or online setting, is a rational metareasoning decision (Russell and Wefald 1991; Hay et al. 2012). Making this decision in an informed and systematic way is another topic for research.

References

Bar-Noy, A., and Schieber, B. 1991. The Canadian traveller problem. In *Proc. of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '91, 261–270.

Becker, A. M., and Ziff, R. M. 2009. Percolation thresholds on two-dimensional voronoi networks and delaunay triangulations. *Phys. Rev. E* 80:041101.

Bnaya, Z.; Felner, A.; and Shimony, S. E. 2009. Canadian traveler problem with remote sensing. In Boutilier, C., ed., *IJCAI*, 437–442.

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

Browne, C.; Powley, E.; Whitehouse, D.; Lucas, S.; Cowling, P.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of Monte Carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on* 4(1):1–43.

Eyerich, P.; Keller, T.; and Helmert, M. 2010. High-quality policies for the Canadian traveler's problem. In *Proc. of the Twenty-Fourth AAAI Conference on Artificial Intelligence, Atlanta, Georgia, USA, July 11-15, 2010*.

Fried, D.; Shimony, S. E.; Benbassat, A.; and Wenner, C. 2013. Complexity of Canadian traveler problem variants. *Theor. Comput. Sci.* 487:1–16.

Goodman, N. D.; Mansinghka, V. K.; Roy, D. M.; Bonawitz, K.; and Tenenbaum, J. B. 2008. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*.

Gordon, A. D.; Henzinger, T. A.; Nori, A. V.; and Rajamani, S. K. 2014. Probabilistic programming. In *International Conference on Software Engineering (ICSE, FOSE track)*.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. 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*, 1163–1168.

Hay, N.; Russell, S. J.; Tolpin, D.; and Shimony, S. E. 2012. Selecting computations: Theory and applications. In *UAI*, 346–355.

Helmert, M. 2006. The Fast Downward planning system. *J. Artif. Int. Res.* 26(1):191–246.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte Carlo planning. In *Proc. of the 17th European Conference on Machine Learning*, ECML'06, 282–293.

Lauritzen, S. 1996. *Graphical Models*. Clarendon Press.

Mansinghka, V. K.; Selsam, D.; and Perov, Y. N. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR* abs/1404.0099.

Nikolova, E., and Karger, D. R. 2008. Route planning under uncertainty: The Canadian traveller problem. In *Proc. of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, 969–974. AAAI Press.

Paige, B.; Wood, F.; Doucet, A.; and Teh, Y. 2014. Asynchronous anytime sequential Monte Carlo. In *Advances in Neural Information Processing Systems*.

Papadimitriou, C. H., and Yannakakis, M. 1991. Shortest paths without a map. *Theor. Comput. Sci.* 84(1):127–150.

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

Russell, S., and Wefald, E. 1991. *Do the right thing: studies in limited rationality*. Cambridge, MA, USA: MIT Press.

Shimony, S. E., and Charniak, E. 1991. A new algorithm for finding MAP assignments to belief networks. In *Proc. of the Sixth Annual Conference on Uncertainty in Artificial Intelligence*, UAI '90, 185–196. New York, NY, USA: Elsevier Science Inc.

Sun, X.; Druzdzel, M. J.; and Yuan, C. 2007. Dynamic weighting A* search-based MAP algorithm for bayesian networks. In *Proc. of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, 2385–2390. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Sutton, R. S., and Barto, A. G. 1998. *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1st edition.

Szörényi, B.; Kedenburg, G.; and Munos, R. 2014. Optimistic planning in markov decision processes using a generative model. In Ghahramani, Z.; Welling, M.; Cortes, C.; Lawrence, N.; and Weinberger, K., eds., *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc. 1035–1043.

Wingate, D.; Stuhlmüller, A.; and Goodman, N. D. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proc. of the 14th Artificial Intelligence and Statistics*.

Wood, F.; van de Meent, J. W.; and Mansinghka, V. 2014. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*.

Automatic Generation of HTNs From PDDL

Anders Jonsson
Universitat Pompeu Fabra
08018 Barcelona, Spain
anders.jonsson@upf.edu

Damir Lotinac
Universitat Pompeu Fabra
08018 Barcelona, Spain
damir.lotinac@upf.edu

Abstract

Hierarchical Task Networks (HTNs) are a common model for encoding knowledge about planning domains in the form of task decompositions. Since an HTN is parameterized it can be used to solve any instance of a planning domain. Thus HTN planning is an interesting candidate for generalizing knowledge about a planning instance to other instances of the same domain, just like in the learning track of the International Planning Competition.

We present a novel algorithm that automatically generates an HTN from the PDDL description of a planning domain and a single representative instance. The HTNs that our algorithm constructs contain two types of composite tasks that interact to achieve the goal of a planning instance. One type of task achieves fluents by traversing the edges of invariant graphs in which only one fluent can be true at a time. The other type of task traverses a single edge of an invariant graph by applying the associated action, which first involves ensuring that the preconditions of the action hold. The resulting HTNs can be applied to any instance of a planning domain, and are provably sound, such that the solution to an HTN instance can always be translated back to a solution to the original planning instance. In several domains we are able to solve most or all planning instances using HTNs created from a single example instance.

Introduction

Hierarchical Task Networks, or HTNs, are a popular tool for encoding hierarchical structure into planning domains. In the past, HTNs have been successfully used in a variety of planning applications: military planning (Munoz-Avila et al. 1999), Web service composition (Wu et al. 2003), unmanned air vehicle control (Miller et al. 2004), strategic game playing (van der Sterren 2009; Menif, Guettier, and Cazenave 2013), personalized patient care (Sánchez-Garzón, Fernández-Olivares, and Castillo 2013) and business process management (González-Ferrer, Fernández-Olivares, and Castillo 2013), to name a few.

Although HTNs are known to be at least as expressive as STRIPS planning (Erol, Hendler, and Nau 1994), this is not the main reason for their popularity. Quite the opposite, in fact. Arguably, the most important knowledge encoded in an HTN is not the possible ways to expand its tasks, but rather the ways in which the tasks *cannot* be expanded. By excluding portions of the state space, search proceeds more quickly towards the goal, or in HTN terminology, towards generating a valid expansion of the initial task list. In the extreme case, each task has a single possible expansion, and planning is reduced to a simple traversal of the task hierarchy.

Another powerful characteristic of HTNs is that tasks are parameterized, making it possible to encode knowledge about an entire planning domain, not just individual planning instances. Although identifying effective decomposition strategies for all instances of a domain can be arduous, once an HTN has been constructed for a planning domain, it can be used to solve an entire family of instances more efficiently.

Thus, the main reason that HTNs are frequently used in real-world applications is that they offer a potent mechanism for reducing the search effort required to solve a family of large-scale planning instances. This is also the reason that HTNs were so successful in the hand tailored track of early planning competitions (IPC): the participants were given access to the planning domains beforehand and designed HTNs that effectively narrowed the search to a tiny portion of the state space. It is not a coincidence that the HTN planner that achieved the largest coverage at IPC-2002 and is widely regarded as the state-of-the-art in HTN planning, SHOP2 (Nau et al. 2003), performs blind search in the task space to compute a valid expansion. Most of the work required to reduce the search effort is performed while designing the HTN, and once this work is done, there is little need to optimize search.

In summary, HTNs offer a mechanism for human experts to encode prior knowledge about a planning domain. Typically, this requires many hours of fine-tuning, debugging and testing. This is fine for planning applications in which the initial effort is compensated by the subsequent reduction in search time during successive applications of the planner. However, a large body of research in the planning community is dedicated to finding domain-independent approaches to planning. Traditionally, HTNs have not found a place in this research because of their domain-dependent emphasis.

In this paper we ask the following question: is it possible to devise domain-independent strategies for automatically deriving HTNs that offer some of the same benefits

as those designed by human experts? Although there have been earlier attempts to learn HTNs automatically (Hogg, Munoz-Avila, and Kuter 2008; Zhuo et al. 2009), these approaches rely on partial information about the task decomposition, which already encodes much of the domain knowledge required to define HTNs. In contrast, our approach is to generate HTNs directly from the PDDL encoding of a planning domain and a single representative instance.

Our approach is to generate HTNs that encode invariant graphs of planning domains. An invariant graph is similar to a lifted domain transition graph, but can be subdivided on types. To traverse an invariant graph we define two types of tasks: one that reaches a certain node of an invariant graph, achieving the associated fluent, and one that traverses a single edge of an invariant graph, applying the associated action. These two types of tasks are interleaved, in that the expansion of one type of task involves tasks of the other type.

In experiments, we tested our approach on planning benchmarks from the IPC. In four domains, our algorithm is able to construct HTNs that make it possible to efficiently solve any instance of the domain using blind search. The approach is partially successful in other domains, but the branching factor becomes a problem for large instances. Still, the results indicate that automatically generating HTNs is possible.

The rest of the paper is organized as follows. We first provide a background on planning and HTNs. We then introduce the concept of invariant graphs, and describe our base algorithm for constructing HTNs. Next, we describe several optimizations on top of the base algorithm, and present experimental results. We conclude with a discussion of related work and possible directions for future work.

Planning

In this section we introduce notation related to planning and HTNs. Since our work depends heavily on lifted representations, we use PDDL notation to define actions and tasks.

Planning Domains

We consider the fragment of PDDL modelling typed STRIPS planning domains with positive preconditions and goals. A planning domain is a tuple $\boldsymbol{d} = \langle \mathcal{T}, \prec, P, A \rangle$, where $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ is a set of types, $\prec$ an inheritance relation on types, P a set of predicates and A a set of actions. Each predicate $p \in P$ and action $a \in A$ has a parameter list ($\varphi(p)$ and $\varphi(a)$, respectively) whose elements are types. For example, the types and predicates in LOGISTICS are

```
(:types truck airplane - vehicle
        package vehicle - physobj
        airport location - place
        city place physobj - object)
(:predicates (incity ?loc - place ?city - city)
             (at ?obj - physobj ?loc - place)
             (in ?pkg - package ?veh - vehicle)).
```

Each action $a \in A$ has a precondition $\mathrm{pre}(a)$, an add effect $\mathrm{add}(a)$, and a delete effect $\mathrm{del}(a)$. Each precondition and effect consists of a predicate p and a mapping from $\varphi(p)$ to $\varphi(a)$. An example action from LOGISTICS is given by

```
(:action loadtruck
 :parameters (?p - pkg ?t - truck ?l - place)
 :precondition (and (at ?t ?l) (at ?p ?l))
 :effect  (and (not (at ?p ?l)) (in ?p ?t))).
```

Note that the parameters of all preconditions and effects refer to the parameters of the action `loadtruck`.

Given a domain $\boldsymbol{d}$, a STRIPS planning instance is a tuple $\boldsymbol{p} = \langle \Omega, \mathrm{init}, \mathrm{goal} \rangle$, where $\Omega = \Omega_1 \cup \ldots \cup \Omega_n$ is a set of objects of each type, init is an initial state, and goal is a goal state. Object $\omega \in \Omega$ has type $\tau \in \mathcal{T}$ iff $\omega \in \Omega_i$ for some $1 \leq i \leq n$ and τ_i equals τ or inherits from τ. Instance $\boldsymbol{p}$ implicitly defines a set of fluents $F = \{(p\ v_1 \cdots v_{|\varphi(p)|}) \mid p \in P, v_i \in \Omega_{\varphi(p)_i}, 1 \leq i \leq |\varphi(p)|\}$ consisting of assignments of objects in Ω to the parameters of predicates such that each object has the appropriate type. The initial state $\mathrm{init} \subseteq F$ and goal state $\mathrm{goal} \subseteq F$ are both subsets of fluents.

The planning instance also defines a set of operators $O = \{(a\ v_1 \cdots v_{|\varphi(a)|}) \mid a \in A, v_i \in \Omega_{\varphi(a)_i}, 1 \leq i \leq |\varphi(a)|\}$ consisting of assignments of objects in Ω to the parameters of actions. Each operator $o \in O$, $o = (a\ v_1 \cdots v_{|\varphi(a)|})$, has a precondtion $\mathrm{pre}(o) \subseteq F$, and add effect $\mathrm{add}(o) \subseteq F$ and a delete effect $\mathrm{del}(o) \subseteq F$, each a subset of fluents instantiated from the preconditions and effects of a by copying objects among $v_1, \ldots, v_{|\varphi(a)|}$ onto the parameters of predicates. For example, the operator (`loadtruck p1 t1 l2`) in LOGISTICS has preconditions (`at t1 l2`) and (`at p1 l2`).

A state $s \subseteq F$ is a subset of fluents that are true, while fluents in $F \setminus s$ are false. An operator $o \in O$ is applicable in s if and only if $\mathrm{pre}(o) \subseteq s$, and the result of applying o in s is a new state $s \ltimes o = (s \setminus \mathrm{del}(o)) \cup \mathrm{add}(o)$. A plan for $\boldsymbol{p}$ is a sequence of operators $\pi = \langle o_1, \ldots, o_n \rangle$ such that o_i, $1 \leq i \leq n$, is applicable in $I \ltimes o_1 \ltimes \cdots \ltimes o_{i-1}$, and π solves $\boldsymbol{p}$ if it reaches the goal state, i.e. if $G \subseteq I \ltimes o_1 \ltimes \cdots \ltimes o_n$.

We use this LOGISTICS instance as a running example:

```
(define (problem logistics-example)
(:domain logistics)
(:objects a1 - airplane ap1 ap2 - airport
          c1 c2 - city l1 l2 - location
          t1 t2 - truck p1 - package)
(:init (at p1 l1) (at a1 ap2)
       (at t1 l1) (incity l1 c1) (incity ap1 c1)
       (at t2 l2) (incity l2 c2) (incity ap2 c2))
(:goal (and (at p1 ap1)))).
```

Hierarchical Task Networks

We use a notation inspired by Geier and Bercher (2011) to describe HTNs. Their notation, however, is grounded, so we introduce a lifted representation similar to STRIPS domains. In this context, an HTN domain is a tuple $\boldsymbol{h} = \langle P, A, C, M \rangle$, where P is a set of predicates, A is a set of actions (i.e. primitive tasks), C is a set of compound tasks and M is a set of decomposition methods. Each task $c \in C$ and method $m \in M$ has an associated parameter list $\varphi(c)$ and $\varphi(m)$, respectively. Unlike STRIPS domains, HTN domains are untyped and we allow negative preconditions.

A *task network* is a tuple $tn = \langle T, \prec \rangle$, where $T \subseteq A \cup C$

is a set of tasks and $\prec$ is a partial order on T. A method $m = \langle c, tn_m, \mathrm{pre}(m)\rangle$ consists of a compound task $c \in C$, a task network $tn_m = \langle T_m, \prec_m\rangle$ and a precondition $\mathrm{pre}(m)$. Each precondition $p \in P$ and task $t \in T_m$ has an associated mapping from $\varphi(p)$ or $\varphi(t)$ to $\varphi(m)$.

Given $\boldsymbol{h} = \langle P, A, C, M\rangle$, an HTN instance is a tuple $\boldsymbol{s} = \langle \Omega, I, tn_I\rangle$, where Ω is a set of objects, I is an initial state and tn_I is a task network. Just like for STRIPS, Ω induces sets F and O of fluents and operators, as well as sets $\mathcal{C}$ and $\mathcal{M}$ of grounded compound tasks and methods. A grounded task network has tasks in $O \cup \mathcal{C}$, and is *primitive* if all tasks are in O. The initial state $I \subseteq F$ is a subset of fluents, and the initial grounded task network $tn_I = \langle\{t_I\}, \emptyset\rangle$ has a single grounded compound task $t_I \in \mathcal{C}$.

We use $(s, tn) \rightarrow_D (s', tn')$ to denote that a pair of a state and a task network decomposes into another pair, where $tn = \langle T, \prec\rangle$ and $tn' = \langle T', \prec'\rangle$. A valid decomposition consists in choosing a task $t \in T$ such that $t' \not\prec t$ for each $t' \in T$, and applying one of the following rules:

1. If t is primitive, the decomposition is applicable if $\mathrm{pre}(t) \subseteq s$, and the resulting pair is given by $s' = s \ltimes t$, $T' = T \setminus \{t\}$ and $\prec' = \{(t_1, t_2) \in \prec \mid t_1, t_2 \in T'\}$.
2. If t is compound, the decomposition method $m = \langle t, tn_m, \mathrm{pre}(m)\rangle$ is applicable if $\mathrm{pre}(m) \subseteq s$, and the resulting pair is given by $s' = s, T' = T \setminus \{t\} \cup T_m$ and

$$\begin{aligned}\prec' = {} & \{(t_1, t_2) \in \prec \mid t_1, t_2 \in T'\} \\ & \cup \{(t_1, t_2) \in T' \times T_m \mid (t_1, t) \in \prec\} \\ & \cup \{(t_2, t_1) \in T_m \times T' \mid (t, t_1) \in \prec\}.\end{aligned}$$

The first rule removes a primitive task t from tn and applies the effects of t to the current state, while the second rule uses a method m to replace a compound task t with tn_m while leaving the state unchanged. If there is a finite sequence of decompositions from (s_1, tn_1) to (s_n, tn_n) we write $(s_1, tn_1) \rightarrow_D^* (s_n, tn_n)$. An HTN instance is solvable if and only if $(s_I, tn_I) \rightarrow_D^* (s_n, \langle\emptyset, \emptyset\rangle)$ for some state s_n, i.e. the resulting task network is empty.

Invariants

In STRIPS planning, a mutex invariant is a subset of fluents such that at most one is true at any moment. An algorithm for detecting mutex invariants is implemented as part of the Fast Downward planning system (Helmert 2009). This algorithm is of particular interest to us since it is instance-independent: the algorithm only relies on the domain description to detect invariants. Unlike Fast Downward, which later grounds the resulting invariants in a specific planning instance, our algorithm operates directly on the invariants extracted from the domain description.

In LOGISTICS, Fast Downward finds a single invariant $\{$(in ?o ?v),(at ?o ?p)$\}$, i.e. a set of predicates with associated variable lists. Variables that appear in the lists of all predicates are *bound*, i.e. take on the same value for all predicates. The remaining variables are *free* and can be replaced with any object of the given type. The meaning of the invariant is that across all LOGISTICS instances, a given object ?o is either in a vehicle or at a location.

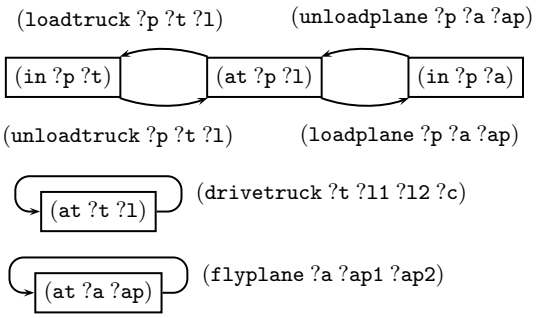


Figure 1: Invariant graphs in LOGISTICS.

An invariant can be grounded on a concrete assignment of objects from a planning instance. In our running example, assigning the package p1 to the bound variable ?o in the invariant above and considering all assignments of objects to the free variables results in the grounded invariant

$\{$(at p1 ap1),(at p1 ap2),(at p1 l1),(at p1 l2),
(in p1 t1),(in p1 t2),(in p1 a1)$\}$.

If a predicate $p \in P$ is not part of any invariant but there are actions that add and/or delete p, we create a new invariant $\{(p\ \ ?o1 \cdots\ ?ok),(\neg p\ \ ?o1 \cdots\ ?ok))\}$, i.e. all variables are bound and an associated fluent can either be true or false.

Given an invariant, our algorithm generates one or several invariant graphs. In LOGISTICS, all actions affect the lone invariant above. However, when loading or unloading a package, the bound object ?o is a package, when driving a truck ?o is a truck, and when flying an airplane ?o is an airplane. Moreover, we can either load a package into a truck or an airplane. We differentiate between types such that each invariant may correspond to multiple invariant graphs.

To generate the invariant graphs we go through each action, find each transition of each invariant that it induces (by pairing add and delete effects and testing whether the bound objects are identical), and map the types of the predicates to the invariant. We then either create a new invariant graph for the bound types or add nodes to an existing graph corresponding to the mapped predicate parameters.

Figure 1 shows the invariant graphs in LOGISTICS. In the top graph (G_1), the bound object is a package ?p, in the middle graph (G_2) a truck ?t, and in the bottom graph (G_3) an airplane ?a. Note that the predicate in is not actually part of the two bottom graphs, since trucks and planes cannot be inside other vehicles. Nevertheless, the invariant still applies: a truck or plane can only be at a single place at once.

Each edge of an invariant graph corresponds to an action that deletes one predicate of the invariant and adds another. To do so, the parameters of the action have to include the parameters of both predicates, including the bound objects. In the figure, the invariant notation is extended to actions on edges such that each parameter is either bound or free.

Even if the actions of the domain preserve the invariant property, the initial state of a planning instance may contain more than one fluent of an associated grounded invariant, in

which case the grounded invariant is not a mutex invariant. To verify that an invariant corresponds to actual grounded invariants, our algorithm needs access to the initial state of an example planning instance $\boldsymbol{p}$ of the domain. If this verification fails, the invariant is not considered by the algorithm.

Generating HTNs

In this section we describe our algorithm for automatically generating HTNs. The idea is to construct a hierarchy of tasks that traverse the invariant graphs to achieve certain fluents. In doing so there are two types of interleaved tasks: one that achieves a fluent in a given invariant (which involves applying a series of actions to traverse the edges of the graph), and one that applies the action on a given edge (which involves achieving the preconditions of the action).

Formally, our algorithm takes as input a STRIPS planning domain $\boldsymbol{d} = \langle \mathcal{T}, \prec, P, A \rangle$ and outputs an HTN domain $\boldsymbol{h} = \langle P', A', C, M \rangle$. The algorithm first constructs the invariant graphs $G_1, \ldots, G_k$ described above. Below we describe the components of the generated HTN domain $\boldsymbol{h}$.

Predicates

The set $P' \supseteq P$ extends P with three predicates for each $p \in P$: `persist-`p, `visited-`p, and `achieving-`p. Respectively we use these predicates to temporarily cause p to persist, to flag p as an already visited node during search, and to prevent infinite recursion in case p or another predicate from the same invariant is currently being achieved.

Tasks

Each action $a \in A$ from the input STRIPS planning domain $\boldsymbol{d}$ becomes a primitive task of $\boldsymbol{h}$. We add extra preconditions to ensure that a is not grounded on the wrong type, and that a does not delete a predicate that is supposed to persist. We also add primitive tasks for visiting, locking and unlocking each predicate $p \in P$. Visiting marks a predicate as visited, locking causes a predicate to temporarily persist, while unlocking frees a predicate so that it can be deleted again. For each invariant graph G_i, we add two primitive tasks `set-flags-`i that marks each predicate $p \in P$ in G_i as being achieved, and `clear-flags-`i that clears all flags for G_i.

We also include three types of compound tasks:

- For each predicate $p \in P$ that appears as positive in any invariant graph, a task `achieve-`p.
- For each invariant graph G_i and each $p \in P$ that is positive in G_i, a task `achieve-`p-i.
- For each invariant graph G_i, each predicate $p \in P$ in G_i, and each outgoing edge of p (corresponding to an action $a \in A$), a task `do-`p-a-i.

The first task is a wrapper task that achieves a predicate p in any invariant, while the other two are the interleaved tasks for achieving p by traversing the edges of an invariant graph G_i. Since the preconditions and goals of the planning domain are positive, we never have to achieve a negated fluent.

Methods

We describe the methods associated with each of the three types of compound tasks in turn. We outline methods in pseudo-SHOP2 syntax, with parameters omitted, in the following format:

(:method (⟨name⟩)
 (⟨precondition⟩)
 (⟨tasklist⟩)).

The first type of task, `achieve-`p, has one associated method for each invariant graph G_i in which p appears. An outline of this method is given by

(:method (`achieve-`p)
 ((¬`achieving-`p))
 ((`set-flags-`i) (`achieve-`p-i) (`lock-`p) (`clear-flags-`i))).

Intuitively this method guides the search to those invariant graphs where p can be achieved. This corresponds to the compound task `achieve-`p-i for some invariant graph G_i. The method also sets and clears the flags of predicates in G_i to prevent infinite recursion, and temporarily locks p to prevent p from being deleted.

The second type of compound task, `achieve-`p-i, has one associated method for each predicate p' in the invariant graph G_i and each outgoing edge of p' (corresponding to an action a):

(:method (`achieve-`p-i)
 ((p') (¬`visited-`p'))
 ((`visit-`p') (`do-`p'-a-i) (`achieve-`p-i))).

Action a appears on an outgoing edge from p', i.e. a deletes p'. Intuitively, one way to achieve p in G_i, given that we are currently at some different node p', is to traverse the edge associated with a using the compound task `do-`p'-a-i. Before doing so we mark p' as visited to prevent us from visiting p' again. After traversing the edge we recursively achieve p from the resulting node. To stop the recursion we define a "base case", which is a method with the same name (`achieve-`p-i), which is applicable only when p already holds and has no associated compound or primitive tasks.

The third type of compound task, `do-`p-a-i, has only one associated method. The aim is to apply action a to traverse an outgoing edge of p in the invariant graph G_i. To do so, the task list has to ensure that all preconditions $p_1, \ldots, p_k$ of a hold (excluding p, which holds by definition, as well as any static preconditions of a). We define the method as

(:method (`do-`p-a-i)
 ()
 (((`achieve-`p_1) ⋯ (`achieve-`p_k)) (a)
  ((`unlock-`p_1) ⋯ (`unlock-`p_k)))).

The decomposition thus achieves all preconditions of a (locking them temporarily), then applies a, and finally unlocks the preconditions (allowing them to be deleted again).

To restrict the choices when traversing the HTN, we impose a total order on all task lists of methods, except tasks (`achieve-`p_1) ⋯ (`achieve-`p_k) of the method `do-`p-a-i, since it may be difficult to determine in which order to achieve the preconditions of an action.

(achieve-at p1 ap1)	(set-flags-1 p1)	(set-flags-1 p1)	(set-flags-1 p1)	(set-flags-1 p1)
	(achieve-at-1 p1 ap1)	(visit-at p1 l1)	(visit-at p1 l1)	(visit-at p1 l1)
	(lock-at p1 ap1)	(do-at-loadtruck-1 p1 t1 l1)	(achieve-at t1 l1)	(set-flags-2 t1)
	(clear-flags-1 p1)	(achieve-at-1 p1 ap1)	(loadtruck p1 t1 l1)	(achieve-at-2 t1 l1)
		(lock-at p1 ap1)	(unlock-at t1 l1)	(lock-at t1 l1)
		(clear-flags-1 p1)	(achieve-at-1 p1 ap1)	(clear-flags-2 t1)
			(lock-at p1 ap1)	(loadtruck p1 t1 l1)
			(clear-flags-1 p1)	(unlock-at t1 l1)
				(achieve-at-1 p1 ap1)
				(lock-at p1 ap1)
				(clear-flags-1 p1)

Table 1: The first four task expansions of the HTN instance generated from the running example in LOGISTICS.

Planning Instances

Once we have constructed the HTN $\boldsymbol{h}$ we can apply it to any instance of the domain. Given a STRIPS instance $\boldsymbol{p} = \langle \Omega, \text{init}, \text{goal} \rangle$, we construct an HTN instance $\boldsymbol{s} = \langle \Omega, \text{init}', \mathcal{L} \rangle$ as follows. The set of objects of $\boldsymbol{s}$ is identical to that of $\boldsymbol{p}$. The initial state init' includes all fluents in init. Moreover, for each type $\tau \in \mathcal{T}$ of the planning domain and each object $\omega \in \Omega$, init' includes the fluent $(\tau\ \omega)$ if ω has type τ. Given $\text{goal} = \{(p_1), \ldots, (p_k)\}$, the task list is $\mathcal{L} = \langle (\texttt{achieve-}p_1), \ldots, (\texttt{achieve-}p_k) \rangle$, where we do not impose any order on tasks.

We show that the HTN translation is sound. The translation is not always complete; in the future we want to identify subclasses of planning instances for which it is provably complete.

Theorem 1 *Let $\pi = \langle o_1, \ldots, o_m \rangle$ be a solution to $\boldsymbol{s}$, and let π' equal π with all* lock-p, unlock-p, visit-p, achieve-in-i *and* clear-flags-i *operators removed. Then π' is a solution to $\boldsymbol{p}$.*

Proof sketch For π to be a solution to $\boldsymbol{s}$, the precondition of each operator o_i has to hold following the application of $o_1, \ldots, o_{i-1}$. Let us restrict attention to fluents associated with the predicates of the original planning domain. The removed operators have no effect on these fluents, while the remaining operators in π' have the same preconditions and effects on these fluents in $\boldsymbol{p}$ and $\boldsymbol{s}$. Since the initial state on these fluents is the same in $\boldsymbol{p}$ and $\boldsymbol{s}$, π' is a plan for $\boldsymbol{p}$.

Each expansion of $(\texttt{achieve-}p_j)$ applies the operator $(\texttt{lock-}p_j)$ after (p_j) is achieved, causing it to persist, i.e. no operator can delete it. Thus the fluents $(p_1), \ldots, (p_k)$ hold after applying π in $\boldsymbol{s}$, implying that they hold after applying π' in init, satisfying the goal state of $\boldsymbol{p}$.

There is a small technicality that we need to resolve: if (p_j) is a precondition of some action a, it can be unlocked as a result of expanding a grounded task of type do-p-a-i, and subsequently deleted. We solve this by checking whether a precondition already holds and only unlocking it if it did not already hold prior to expanding the grounded task; the details are omitted in the description of the HTN.

Example

In LOGISTICS, our algorithm generates two wrapper tasks achieve-in and achieve-at, and four tasks achieve-in-1, achieve-at-1, achieve-at-2, and achieve-at-3, corresponding to the predicates in invariant graphs. The task achieve-at-1 has five associated methods: one for each edge of the graph G_1, plus the base case method.

Moreover, the algorithm generates six tasks do-at-loadtruck-1, do-at-loadplane-1, do-at-unloadtruck-1, do-at-unloadplane-1, do-at-drivetruck-2, and do-at-flyplane-3, corresponding to the six edges of the graphs. The latter two do not have preconditions besides at (the predicate incity in the precondition of drivetruck is static). The remaining four tasks each achieve a single precondition: the truck or plane being at the associated place.

To illustrate the tasks and associated methods we sketch the task expansions of the HTN instance generated from our running example. The only goal is (at p1 ap1), so the task list $\mathcal{L}$ contains a single task (achieve-at p1 ap1). Table 1 shows the first four task expansions of the HTN instance. In each case, the task to be decomposed is underlined.

The first decomposition is produced by the lone method for (achieve-at p1 ap1). The current node associated with p1 in G_1 is (at p1 l1), with two outgoing edges, corresponding to actions loadtruck and loadplane. Applying the method for (achieve-at-1 p1 ap1) associated with (at p1 l1) and loadtruck produces the second expansion. The only method for (do-at-loadtruck-1 p1 t1 l1) expands to (achieve-at t1 l1), which in turn expands to (achieve-at-2 t1 l1) (the last expansion shown).

Since (at t1 l1) already holds, we can apply the base case method for (achieve-at-2 t1 l1), at which point the only composite task is (achieve-at-1 p1 ap1), recursively achieving (at p1 ap1) from the current node (in p1 t1). Attempting to unload p1 at l1 fails since (at p1 l1) has already been visited. Instead, the only option is to unload p1 at ap1, which achieves the goal (at p1 ap1).

Optimizations

In this section we discuss several optimizations of the base algorithm for generating HTNs.

Ordering Preconditions

Achieving the preconditions of an action a in any order is inefficient since an algorithm solving the HTN instance may have to backtrack repeatedly to find a correct order. For this reason, we include an extension of our algorithm that uses

```
1: function ORDER(a, p)
2:     V ← pre(a) \ {p}, Z ← ⟨⟩
3:     repeat
4:         for p′ ∈ V do
5:             W ← {p} ∪ V \ {p′}
6:             for each invariant graph G_j containing p′ do
7:                 Generate all acyclic paths in G_j to p′
8:                 Test if paths applicable when W persists
9:             end for
10:            if each path achieving p′ is applicable then
11:                V ← V \ {p′}
12:                Z ← ⟨p′, Z⟩
13:            end if
14:        end for
15:    until V, Z converge
16:    return (V, Z)
17: end function
```

Figure 2: Algorithm ordering preconditions of a except p.

a simple inference technique to compute a partial order in which to achieve the preconditions of a.

We define a set of predicates whose value is supposed to persist, and check whether a path through an invariant graph is applicable given these persisting predicates. While doing so, only the values of bound variables are known, while free variables can take on any value. Matching the bound variables of predicates and actions enables us to determine whether an action allows a predicate to persist.

Consider a task of type do-p-a-i, i.e. using action a to delete p. Figure 2 shows how to order all preconditions of a except p. In the algorithm, V is the set of preconditions to be ordered, while Z is a sequence of preconditions, initially empty. The algorithm considers one precondition $p' \in V$ at a time and checks if we can achieve p' while all remaining preconditions persist. If so, we remove p' from V and place it first in Z. We then iterate until no more preconditions can be removed from V, and return (V, Z). In the method m associated with do-p-a-i, the preconditions in Z can be achieved in order. On the other hand, we cannot say anything about the order of preconditions that remain in V.

Goal Ordering

Just as for preconditions, achieving the goals in any order results in more backtracking. To order the goals we implement an algorithm similar to the one for ordering preconditions. While the ordered preconditions are coded into the HTN, the goals are different for each instance of the domain. Since HTNs are instance-independent, our approach is to define new tasks that compute a goal ordering as a preprocessing step.

To accomplish this, we first order the goals of the representative instance passed to the algorithm. We run the precondition ordering algorithm on the set of goal predicates $P_G \subseteq P$, i.e. predicates whose associated fluents appear in the goal. Given an ordering of the predicates in P_G, we then order the set of fluents of each predicate $p \in P_G$ using a similar algorithm. To do so, the invariant graphs need to be partially grounded on each pair of fluents to be ordered.

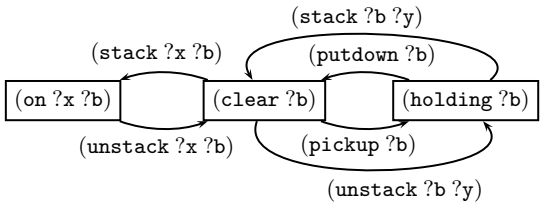


Figure 3: Invariant graph in the BLOCKS domain.

For each pair of fluents $(p\ u_1 \cdots u_k)$ and $(p\ v_1 \cdots v_k)$, we check if $(p\ v_1 \cdots v_k)$ is achievable when $(p\ u_1 \cdots u_k)$ is fixed. Each invariant graph that contains p is partially grounded on $(p\ u_1 \cdots u_k)$, while the preconditions of actions that directly achieve p are grounded on $(p\ v_1 \cdots v_k)$. If this grounding violates the invariant, $(p\ v_1 \cdots v_k)$ should be ordered before $(p\ u_1 \cdots u_k)$. Once the invariant is invalidated by partial grounding, the algorithm stores the indices of the parameters of p that invalidated the invariant.

We introduce two new tasks for goal ordering:

- A new root task `solve`, with an associated method that decomposes to `achieve`-p tasks to achieve predicates in P_G in the order inferred from the representative instance. This method is recursively called for each goal fluent.
- A task `order`, with an associated method that uses the stored parameter indices to order the goal fluents of a single predicate $p \in P_G$.

As an example, in the BLOCKS domain, $P_G = \{\texttt{on}\}$, so the method for `solve` always decomposes to `achieve-on`. Figure 3 shows one of the invariant graphs in BLOCKS that contains `on`. To define the method for `order` we test each pair of goal fluents to establish an order among them. Consider two goal fluents (`on a b`) and (`on b c`). If we fix the fluent (`on a b`) and attempt to achieve (`on b c`), the only operator (`stack b c`) that directly achieves (`on b c`) has precondition (`holding b`), which violates the invariant since (`on a b`) is assumed to hold. Thus (`on b c`) should be ordered before (`on a b`). We can generalize this knowledge and derive a rule that whenever two goal fluents of type `on` have the same object as the first and second parameter, respectively, the former should be ordered before the latter. This general rule is coded into the method for `order`.

Results

We ran our algorithm in the STRIPS planning domains from IPC-2000 and IPC-2002. The reason for choosing these domains is that we can directly compare the performance of our automatically generated HTNs with HTNs hand-crafted by human experts. Since hand-coded planners were not allowed to compete in later competitions, there exist no hand-crafted HTNs from those competitions to compare to.

As a reminder, on the one hand we compare to HTNs designed by experts that had previous access to each domain and ample time to define tasks, on the other to a highly optimized heuristic search planner. In contrast, our approach is

to generate HTNs for each domain in a fraction of a second and to solve HTNs using JSHOP (the Java implementation of SHOP2), which uses blind search to compute a valid expansion.

We performed experiments with two versions of our algorithm. The base algorithm that achieves the preconditions and goals in any order was slow in testing, so we activated precondition ordering in both versions. The first version, HTNPrecon, achieves the goals in the order they appear in the PDDL definition. The second version, HTNGoal, implements our goal ordering strategy in addition to precondition ordering.

For each version of our algorithm, we used JSHOP to solve the resulting HTN instances in each domain. For comparison, we also ran Fast Downward (Helmert 2006) in blind search (FDBlind) setting. Since JSHOP applies blind search to solve HTN instances, it would be unfair to compare to a heuristic planner. We used a memory limit of 4GB and a timeout of 1,800 seconds.

Table 2 shows the results in the 9 domains from the two competitions. For each planner we report the number of instances solved and the maximum time taken to solve an instance. For the HTNs, we also report the maximum number of backtracks (in thousands) performed while solving the HTN instances. Since JSHOP performs blind search, the number of backtracks should approximate the number of tasks expanded. For Fast Downward, we instead report the maximum number of expanded nodes (in thousands).

As expected, our goal ordering strategy mainly improves the performance of the algorithm in BLOCKS and DEPOTS, the two domains that are most sensitive to goal ordering. In addition, goal ordering enables us to solve two additional instances in SATELLITE.

Regarding the hand-crafted HTNs, they successfully solve all instances of all domains with little backtracking; however, recall that our algorithm generates HTNs in a fraction of a second, while the hand-crafted HTNs were carefully designed by human experts. We give times and backtrack information for those hand-crafted domains that could run on JSHOP; for the other domains we give only coverage.

Related Work

Our approach to generating HTNs from a single planning instance and using them to solve larger instances of the same planning domain can be viewed as a form of generalized planning, which has received a lot of recent attention, most notably in the form of the learning track of the IPC. One popular approach to generalized planning is to identify macros (Botea et al. 2005; MacGlashan 2010; Muise et al. 2009; Newton et al. 2007), i.e. sequences of operators that frequently appear in the solutions to example instances. Once identified, such macros can then be inserted into the action space of larger instances in order to speed up search. Another approach is to learn reactive policies for planning domains (Khardon 1999; Levine and Humphreys 2003; Martin and Geffner 2000; Yoon, Fern, and Givan 2002). A third approach is to learn domain-specific knowledge in order to improve heuristic estimates computed during search (de la Rosa et al. 2011; Yoon, Fern, and Givan 2008). In contrast to most of these techniques, which are *inductive*, ours is a *generative* approach to generalized planning.

Achieving fluents by traversing the edges of domain transition graphs is the strategy used by DTGPlan (Chen, Huang, and Zhang 2008) and similar algorithms. There also exist other inference techniques that can solve many individual instances backtrack-free (Lipovetzky and Geffner 2009; 2011). The novelty of our approach is the ability to do this in an instance-independent way.

Our work is also related to other approaches to hierarchical planning (Holte et al. 1996; Marthi, Russell, and Wolfe 2008; Elkawkagy et al. 2012), but ours is the only work that we know of that create the hierarchies automatically.

Conclusion

In this paper we have presented what we believe to be the first domain-independent algorithm for generating HTNs. All the algorithm needs is a PDDL description of the planning domain and a single representative instance. In four domains, the algorithm successfully generates HTNs that can be used to efficiently solve any instance, thus being competitive with HTNs designed by human experts and heuristic search algorithms. Although the success of the algorithm is limited in the remaining domains, we believe that there are still many potential benefits.

First of all, one has to recall that to design HTNs, a human expert typically has to spend hours of work fine-tuning an HTN for each domain. In contrast, our algorithm runs in a fraction of a second. In many cases, even though the resulting HTN is not constrained enough, subtasks identified by the algorithm may still be useful. This is the case, for example, in BLOCKS, where the resulting HTN contains tasks and methods for putting a block on top of another block. In such cases, the algorithm can help suggest initial decompositions that can later be refined by a human expert.

In BLOCKS, the main reason that the algorithm fails is that it has no way of inferring that some blocks should be on the table before achieving the goal. It would of course be easy to add this knowledge to the encoding, but the algorithm would no longer be domain-independent. In almost all other cases for which the algorithm is not as successful, the branching factor of the HTN is too large due to the fact that predicates can be achieved in more than one way. Either there are multiple invariant graphs containing the predicate, or we can traverse an invariant graph in multiple ways. In these cases, an HTN planner has to test all possible ways of achieving a predicate before reporting failure, causing an enormous amount of backtracking.

The avenue for future research that we find most promising is to test different restrictions on the invariant graphs. If the representative instance can still be solved under some restriction, the resulting HTN may still be able to solve other instances, and the restriction has the effect of reducing the branching factor. In essence, this mechanism would reduce the number of ways to traverse the invariant graphs.

Another option is to translate the resulting HTNs back to classical planning instead of using an HTN solver (Alford, Kuter, and Nau 2009; Lekavý and Návrat 2007). In this way

	HTNPrecon			HTNGoal			FDBlind			Hand-crafted		
FREECELL[60]	0	-	-	0	-	-	5	228	17834	60	-	-
BLOCKS[35]	0	-	-	12	50.84	6877	18	32.5	7856	35	0.3	0
ROVERS[20]	20	1.7	16	20	2.0	16	6	219	32787	20	1	1
LOGISTICS[80]	80	8.3	75	80	29.2	75	10	1.58	432	80	-	-
DRIVERLOG[20]	7	74.5	5080	8	60.1	4913	7	40.2	3421	20	-	-
ZENOTRAVEL[20]	4	1527.8	194477	6	1453.8	161365	8	162	5994	20	0.2	0
MICONIC[150]	150	0.66	0	150	0.75	0	55	509	75372	150	0.0	0
SATELLITE[20]	18	0.59	1.2	20	1	0.7	6	702	22982	20	-	-
DEPOTS[22]	4	59.73	4655	15	1178.8	50404	4	53.5	6034	22	-	-

Table 2: Results in the IPC-2000 and IPC-2002 domains, with the number of instances of each domain shown in brackets.

we can take advantage of the reduced branching factor offered by the HTNs, and use heuristic search planners to solve the resulting classical planning instances.

References

Alford, R.; Kuter, U.; and Nau, D. 2009. Translating HTNs to PDDL: A Small Amount of Domain Knowledge Can Go a Long Way. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, 1629–1634.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research* 24:581–621.

Chen, Y.; Huang, R.; and Zhang, W. 2008. Fast Planning by Search in Domain Transition Graphs. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, 886–891.

de la Rosa, T.; Jiménez, S.; Fuentetaja, R.; and Borrajo, D. 2011. Scaling up Heuristic Planning with Relational Decision Trees. *Journal of Artificial Intelligence Research* 40:767–813.

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2012. Improving Hierarchical Planning Performance by the Use of Landmarks. In *Proceedings of the 26th National Conference on Artificial Intelligence (AAAI'12)*.

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, 1123–1128.

Geier, T., and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, 1955–1961.

González-Ferrer, A.; Fernández-Olivares, J.; and Castillo, L. 2013. From Business Process Models to Hierarchical Task Network Planning Domains. *Knowledge Engineering Review* 28(2):175–193.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173:503–535.

Hogg, C.; Munoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, 950–956.

Holte, R.; Perez, M.; Zimmer, R.; and MacDonald, A. 1996. Hierarchical A*: Searching Abstraction Hierarchies Efficiently. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, 530–535.

Khardon, R. 1999. Learning Action Strategies for Planning Domains. *Artificial Intelligence* 113(1-2):125–148.

Lekavý, M., and Návrat, P. 2007. Expressivity of STRIPS-Like and HTN-Like Planning. In *Proceedings of the 1st KES Symposium on Agent and Multi-Agent Systems - Technologies and Applications (KES-AMSTA'07)*, 121–130.

Levine, J., and Humphreys, D. 2003. Learning Action Strategies for Planning Domains Using Genetic Programming. In *EvoWorkshops,* volume 2611 of *Lecture Notes in Computer Science*, 684–695.

Lipovetzky, N., and Geffner, H. 2009. Inference and Decomposition in Planning Using Causal Consistent Chains. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*.

Lipovetzky, N., and Geffner, H. 2011. Searching for Plans with Carefully Designed Probes. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS'11)*.

MacGlashan, J. 2010. Hierarchical Skill Learning for High-Level Planning. In *Proceedings of the 24th National Conference on Artificial Intelligence (AAAI'10)*.

Marthi, B.; Russell, S.; and Wolfe, J. 2008. Angelic Hierarchical Planning: Optimal and Online Algorithms. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS'08)*, 222–231.

Martin, M., and Geffner, H. 2000. Learning Generalized Policies in Planning Using Concept Languages. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR'00)*, 667–677.

Menif, A.; Guettier, C.; and Cazenave, T. 2013. Planning and Execution Control Architecture for Infantry Serious Gaming. In *Proceedings of the 3rd International Planning in Games Workshop (PG'13)*, 31–34.

Miller, C.; Goldman, R.; Funk, H.; Wu, P.; and Pate, B. 2004. A Playbook Approach to Variable Autonomy Control: Application for Control of Multiple, Heterogeneous Unmanned Air Vehicles. In *Annual Meeting of the American Helicopter Society*.

Muise, C.; McIlraith, S.; Baier, J.; and Reimer, M. 2009. Exploiting N-Gram Analysis to Predict Operator Sequences. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*.

Munoz-Avila, H.; Aha, D.; Breslow, L.; and Nau, D. 1999. HICAP: An Interactive Case-Based Planning Architecture and its Application to Noncombatant Evacuation Operations. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI'99)*, 870–875.

Nau, D.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research* 20:379–404.

Newton, M. H.; Levine, J.; Fox, M.; and Long, D. 2007. Learning Macro-Actions for Arbitrary Planners and Domains. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS'07)*, 256–263.

Sánchez-Garzón, I.; Fernández-Olivares, J.; and Castillo, L. 2013. An Approach for Representing and Managing Medical Exceptions in Care Pathways Based on Temporal Hierarchical Planning Techniques. In *Process Support and Knowledge Representation in Health Care (ProHealth'12), Lecture Notes in Computer Science 7738*, 168–182.

van der Sterren, W. 2009. Multi-Unit Planning with HTN and A*. In *AIGameDev Paris Game AI Conference*.

Wu, D.; Parsia, B.; Sirin, E.; Hendler, J.; and Nau, D. 2003. Automating DAML-S Web Services Composition Using SHOP2. In *Proceedings of the 2nd International Semantic Web Conference (ISWC'03)*, 195–210.

Yoon, S.; Fern, A.; and Givan, R. 2002. Inductive Policy Selection for First-Order Markov Decision Processes. In *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence (UAI'02)*, 568–576.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning Control Knowledge for Forward Search Planning. *Journal of Machine Learning Research* 9:683–718.

Zhuo, H.; Hu, D.; Hogg, C.; Yang, Q.; and Munoz-Avila, H. 2009. Learning HTN Method Preconditions and Action Models from Partial Observations. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, 1804–1809.