Extended Abstract: Global Optimisation Techniques for Multi-Agent Planning

Toby Davies
<toby.davies@nicta.com.au>

Abstract

We aim to bridge some of the gap between approaches to planning and scheduling. Planning has very flexible and powerful formalisms for modeling complex optimisation problems that are well suited to the ever-changing requirements of industrial problems, however heuristic search, the defacto standard approach to solving planning problems, is not always the best solving strategy for problems on the intersection of planning and scheduling, such as multi-agent planning with shared resources. Historically, compilation techniques from planning problems to a sequence of satisfiability problems have performed well when makespan optimality was required, however as action costs have become ubiquitous in planning benchmarks, SAT's requirement for a tight bound on the number of actions performed has limited its applicability. Equivalently this work can be seen as testing the feasibility of planning as a modelling language for constraint-programming and mixed integer programming solvers. Our initial work using the Golog action language to specify a hybrid branch-and-price algorithm has shown very promising results. The ultimate aim of this thesis is to develop algorithms to allow heterogenous teams of both model-based and programmatically-specified agents to effectively find and prove their optimal joint plan in domains with shared resources, using a succinct interface that does not require explicit knowledge of other agents in the team.

Publications produced

1. Fragment based planning using column generation (Davies et al. 2014).
2. Optimisation and relaxation in the situation calculus (Davies et al. 2015b).
3. Sequencing Operator Counts (Davies et al. 2015a).

Introduction

Multi-agent temporal *Golog* (Kelly and Pearce 2006) and heuristic optimising *Golog* (Blom and Pearce 2010) have been investigated separately. The combination of these features have industrial applications in scheduling problems, and the ability to use domain knowledge to supplement the search for solutions is attractive, however *Golog*'s lack of powerful search algorithms has limited its applicability. We attempt to address this shortcoming for a class of planning/scheduling problems.

One of the key techniques behind our approach is linear programming, in particular duality theory. Linear programming has been used by a number of planning heuristics (Briel et al. 2007) (Coles et al. 2008) (Bonet 2013). However these heuristics have exploited only the primal solutions to the LP, whereas we use both the primal and the dual. Additionally we use the information in a way, that cannot be described as a heuristic in the usual sense.

Preliminaries

The Situation Calculus and Basic Action Theories. The *situation calculus* is a logical language specifically designed for representing and reasoning about dynamically changing worlds (Reiter 2001). All changes to the world are the result of *actions*, which are terms in the language. We denote action variables by lower case letters a, and action terms by α, possibly with subscripts. A possible world history is represented by a term called a *situation*. The constant S_0 is used to denote the initial situation where no actions have yet been performed. Sequences of actions are built using the function symbol do, such that $do(a, s)$ denotes the successor situation resulting from performing action a in situation s. Predicates and functions whose value varies from situation to situation are called *fluents*, and are denoted by symbols taking a situation term as their last argument (e.g., $Holding(x, s)$).

Within the language, one can formulate *basic action theories* that describe how the world changes as the result of the available actions (Reiter 2001). These theories, combined with *Golog*, are more expressive than STRIPS or ADL (Röger and Nebel 2007). Two special fluents are used to define a legal execution: $Poss(a, s)$ is used to state that action a is executable in situation s; and $Conflict(as, s)$ is used to state that the set of actions as may not be executed concurrently.

High-Level Programs. High-level non-deterministic programs can be used to specify complex goals: the goal of a *Golog* program is to find a sequence of actions generated by some path through the program. We use temporal semantics from *MIndiGolog* (Kelly and Pearce 2006) which builds on

ConGolog (Giacomo, Lesperance, and Levesque 2000), and refer to these extensions simply as *Golog*. A *Golog* program δ is defined in terms of the following structures:

α	atomic action
$\varphi?$	test for a condition
$\delta_1; \delta_2$	sequence
while φ **do** δ	while loop
$\delta_1 \vert \delta_2$	nondeterministic branch
$\pi x.\delta$	nondeterministic choice of argument
δ^*	nondeterministic iteration
$\delta_1 \Vert \delta_2$	concurrency

In the above, α is an action term, possibly with parameters, δ is a *Golog* program, and φ a situation-suppressed formula, that is, a formula in the language with all situation arguments in fluents suppressed. Program $\delta_1 | \delta_2$ allows for the nondeterministic choice between programs δ_1 and δ_2, while $\pi x.\delta$ executes program δ for *some* nondeterministic choice of a legal binding for variable x. δ^* performs δ zero or more times. Program $\phi?$ succeeds only if ϕ holds in the current situation. Program $\delta_1 \| \delta_2$ expresses the concurrent execution of programs δ_1 and δ_2. For notational convenience we add:

$\pi(x \in X).\delta$	equivalent to $\pi x.(x \in X)?; \delta$
`foreach` x `in` vs `do` δ	equivalent to $\delta_{[x/v_1]}; \cdots ; \delta_{[x/v_n]}$
`forconc` x `in` vs `do` δ	equivalent to $\delta_{[x/v_1]} \Vert \cdots \Vert \delta_{[x/v_n]}$

Here $\delta_{[x/y]}$ denotes the program δ where each occurrence of variable x has been replaced with the value y, and v_i is the ith element of the sequence vs.

Linear and Integer Programming. An Integer Program (IP) consists of a vector of binary decision variables,[1] usually denoted $\tilde{\boldsymbol{x}}$, an objective linear expression and a set of linear inequalities. We will refer to these inequalities as "resources" throughout this paper, and each decision represented by the variable x_i can be thought of as using $u_{r,i}$ units of some resources $r \in R$, each of which has an availability of a_r.

A general form, assuming a set R of inequalities, where c_i, $u_{r,i}$ and a_r are constants, is:

$$\begin{aligned} \text{Minimize:} \quad & \sum c_i \cdot x_i \\ \text{Subject To:} \quad & \sum u_{r,i} \cdot x_i \quad \leq a_r \qquad \forall r \in R \end{aligned}$$

Finding the optimal solution to an integer program is NP-hard, however the linear program (LP), constructed by replacing the integrality constraints $x_i \in \{0, 1\}$ with a continuous equivalent $x_i \in [0, 1]$, can be optimised in polynomial time. This LP is known as the linear relaxation of the IP.

A model of this form where some variables are continuous and some are integral is called a Mixed Integer Program (MIP). To limit confusion, we will denote binary variables x_i and continuous ones v_i, we assume all $x_i \in \{0, 1\}$ and $v_i \in [0, 1]$ throughout this paper.

[1] In general decision variables can be integer but binary decision variables suffice for our purposes.

The (M)IP is then solved using a "branch-and-bound" search strategy where some x_i which is fractional in the LP optimum is selected at each node and two children are enqueued with additional constraints $x_i = 1$ in one branch and $x_i = 0$ in the other. Heuristically constructed integer solutions provide an upper bound, and the relaxations provide a lower bound.

Solving the linear relaxation implicitly also optimises the so-called dual problem. Intuitively the dual problem is another linear program that seeks to minimize the unit price of each resource in R, subject to the constraint that it must under-estimate the optimal objective of the primal. We use π_r to denote this so-called "dual-price" of resource r. An estimate of the impact of consuming u additional units of resource r on the current objective can then be computed by multiplying usage by the dual price: $u \cdot \pi_r$.

These dual prices allow us to quickly identify bottlenecks in a system, and give us an upper bound on just how far out of the way we should consider going to avoid them. This leads to the important concept of reduced cost: an estimate of how much a decision can improve an incumbent solution. Informally, reduced cost is the decision's intrinsic cost c_i, less the total dual price of the resources required to make this decision.

Given an incumbent dual solution $\tilde{\boldsymbol{\pi}}$, a decision variable x_i has a reduced-cost $\gamma(i, \tilde{\boldsymbol{\pi}})$, defined as:

$$\gamma(i, \tilde{\boldsymbol{\pi}}) = c_i - \sum_{r \in R} \pi_r \cdot u_{r,i}$$

This is guaranteed to be a locally optimistic estimate, so that, in order to improve an incumbent solution, we only need to consider decisions x_i with $\gamma(i, \tilde{\boldsymbol{\pi}}) < 0$. Due to the convexity of linear programs, repeatedly improving an incumbent solution is sufficient to eventually reach the global optimum, and the non-existence of any x_i with negative reduced cost is sufficient to prove global optimality.

Column Generation and Branch-and-Price. Most real-world integer programs have a very small number of non-zero x_i. This property, combined with the need only to consider negative reduced-cost decision variables, allows us to solve problems with otherwise intractably large decision vectors using a process known as "column generation" (Desaulniers, Desrosiers, and Solomon 2005). The name reflects the fact that the new decision variable is an additional column in the matrix representation of the constraints.

Column generation starts with a restricted set of decision variables obtained by some problem-dependent method to yield a linear feasible initial solution. With such a solution we can compute duals for this restricted master problem (RMP) and use reduced-cost reasoning to prune huge areas of the column space.

Incomplete and suboptimal methods of constructing integer feasible solutions are unfortunately referred to in the Operations Research literature simply as "Heuristics", to avoid ambiguity we will refer to them as "MIP heuristics". These are essential for both finding a feasible initial solution, and

providing a worst bound on the solution during the branch-and-bound search process. These are analogues to fast but incomplete search algorithms in planning such as enforced hill climbing.

Column generation then proceeds by repeatedly solving one or more "pricing problems" to generate new decision variables with negative reduced cost and re-solving the RMP to generate new dual prices. Iterating this process until no more negative reduced cost columns exist is guaranteed to reach a fixed point with the same objective value as the much larger original linear program. We can then use a similar "branch-and-bound" approach as in integer programming to reach an integer optimum. This process is known as "branch-and-price".

Branching rules used in practical branch-and-price solvers are often more complex than in IP branch-and-bound and are sometimes problem dependent. The branch $x_i = 0$ does not often partition the search-space evenly: there are usually exponentially many ways to use the resources consumed by x_i. Additionally, disallowing the re-generation of the same specific solution x_i by the pricing problem is not possible with an IP-based pricing-problem without fundamentally changing the structure of the pricing-problem.

Consequently, effectiveness of a branching strategy must be evaluated in terms of how effectively the dual-price of the branching constraint can be integrated into the pricing problem. This is another motivation for our hybrid IP/Planning approach, as using a planning-based pricing problem allows us to disallow specific solutions and handle non-linear, time-dependent costs and constraints.

A concrete example of branch-and-price is presented in the Fragment-Based Planning section.

Big-M Penalty Methods. We noted earlier that to start column generation an initial (linear) feasible solution is required. There is no guarantee that finding such a feasible solution is trivial. Indeed for classical Planning problems, finding a feasible solution is PSPACE-complete in general. No work we are aware of determines the complexity of computing a linear relaxation of a plan.

We avoid this problem by transforming our IP into an MIP where for any constraint having $a_r < 0$ we add a new continuous variable $v_r \in [0, 1]$, representing the degree to which the constraint is violated, and replace the constraint with: $\sum u_{r,i} \cdot x_i \leq |a_r| \cdot v_r - |a_r|$ and $c_v = M$ where M is a large number. This guarantees the feasibility of the trivial solution $x_i = 0$ for all i and all $v_r = 1$.

This represents a relaxation of the original IP with the property that, given sufficiently large M, the optimal solution is a feasible solution to the original problem, iff such a solution exists. This is known as a "penalty method" or "soft constraint". The process is similar to the first phase of the simplex algorithm for finding an initial feasible solution to a linear program when all decision variables are known in advance.

Reasoning about optimality One of the key techniques in proving solution quality used in Operations Research is relaxation. Formally a relaxation of a problem with a set of solutions X and cost function C is a new problem with solutions X' and cost function C' such that $X \subseteq X'$ and for any solution $x \in X$, the relaxed cost is a lower bound on the real cost $C'(x) \leq C(x)$.

Relaxations should be easier to solve to optimality than the original problem, and can give us a tractable way to prove how far from optimal a candidate solution is. Given a feasible solution to the original problem x and an optimal solution to the relaxed problem x'^*, the optimality gap ϵ is defined as $\epsilon = C(x)/C'(x'^*) - 1$. When $\epsilon = 0$ the solution is optimal. Where multiple relaxations are available, the tightest can be used to compute ϵ. This approach is used with Lagrangian Relaxation.

Lagrangian Relaxation In many optimisation problems it is easier to optimize a variant of the problem with fewer constraints. For example the resource constrained shortest path problem (Mehlhorn and Ziegelmann 2000) is NP-complete, whereas there are well-known polynomial-time algorithms, such as Dijkstra's, for the classical, unconstrained shortest path problem.

Lagrangian Relaxation takes advantage of this property by softening such complicating constraints into the objective function (Lemaréchal 2001; Fisher 2004). This allows the effective use of algorithms designed for the easier problem by penalizing violations of the complicating constraints. Careful variation of the penalties (Lagrange multipliers) allows the relaxed problem to be guided towards feasible areas of the original problem.

Traditionally Lagrangian Relaxation is applied to Integer Programming (IP) models. A significant part of the contribution of my thesis is extending this concept to logic-based optimization in dynamical systems.

Given a set of inequalities R to relax in an IP model:

$$\begin{array}{lll} \text{Minimize:} & z(\tilde{\boldsymbol{x}}) & \\ \text{Subject To:} & c_r(\tilde{\boldsymbol{x}}) \leq 0 & \forall r \in R \\ & c_n(\tilde{\boldsymbol{x}}) \leq 0 & \forall n \in N \\ & \tilde{\boldsymbol{x}} \in \mathcal{Z}^n & \end{array}$$

We transform it into a relaxed problem:

$$\begin{array}{lll} \text{Minimize:} & z(\tilde{\boldsymbol{x}}) + \sum_{r \in R} \lambda_r \cdot c_r(\tilde{\boldsymbol{x}}) & \\ \text{Subject To:} & c_n(\tilde{\boldsymbol{x}}) \leq 0 & \forall n \in N \\ & \tilde{\boldsymbol{x}} \in \mathcal{Z}^n & \end{array}$$

Note that this increases the objective when constraints are violated, but decreases it when constraints are strictly satisfied. To solve the original problem, the relaxed problem is optimized and the penalty, λ_r, for each violated constraint, r, is increased, the relaxed problem is re-optimized and the process is iterated. In general, branching is also required to find solutions to discrete problems.

Fragment-Based Planning

Given the set F of all possible executions of each agents' programs, all joint executions for the team are a subset of F where no two fragments use the same resource simultaneously

Finding the optimal execution is then equivalent to finding the optimal solution to the Integer Program:

$$\text{Minimize:} \quad \sum_{f \in F} d_f \cdot x_f + \sum_{o \in O} M \cdot v_o$$

$$\text{Subject To:} \quad \sum_{f \in F_{bt}} x_f \leq c(b) \qquad \forall b \in B\,, \forall t \in T$$

$$v_o + \sum_{f \in F_o} x_f = 1 \qquad \forall o \in O$$

Here x_f is 1 iff f should be executed in the joint plan. $F_o \subseteq F$ is the set of fragments that satisfy subgoal o, $F_{bt} \subseteq F$ is the set of fragments that consume resource b at time t and d_f is the duration of fragment f.

Enumerating F is however prohibitive, and large numbers of potential fragments are uninteresting, or prohibitively costly and will never be chosen in any reasonable joint-execution, nor need to be considered in finding and proving the optimal joint execution.

To avoid enumerating F, we can use delayed column generation as described earlier. To use this approach, we need to re-compute action costs that minimise the reduced cost of the next fragment generated given an optimal solution to the restricted LP.

We provide pseudo-code for the FBP algorithm below, Quine quotes are used around linear expressions such as $[\![expression \leq constant]\!]$ to denote constraints given to the LP solver to distinguish them from logical expressions.

To solve the linear relaxation of the joint planning problem, we call LINFBP$(\{v_o \mid o \in O\}, \{o : [\![v_o = 1]\!] \mid o \in O\}, O, \delta)$. Note that the $[\![expr = a]\!]$ form of constraints can be considered a shorthand for two constraints $[\![expr \leq a]\!]$ and $[\![-expr \leq -a]\!]$.

We assume that the *Golog* search procedure Do returns the fragment f with the least reduced cost $\gamma(f, \tilde{\boldsymbol{\pi}})$, rather than just any valid execution. Our implementation uses uniform-cost search to achieve this.

function LINFBP(*Frags*, *Res*, *Goals*, δ)
  LowBound $\leftarrow 0$
  UpBound $\leftarrow M \cdot \|Goals\|$
  while $(1-\epsilon) \cdot$ *UpBound* $>$ *LowBound* **do**
    $\theta, \tilde{\boldsymbol{x}}, \tilde{\boldsymbol{\pi}} \leftarrow$ *SolveLP*(*Frags*, *Res*)
    $F \leftarrow Do(\pi(g \in Goals).\delta | noop, \tilde{\boldsymbol{\pi}})$
    Frags $\leftarrow$ *Frags* $\cup \{F\}$
    $d\theta \leftarrow \|Goals\| \cdot \gamma(F, \tilde{\boldsymbol{\pi}})$
    UpBound $\leftarrow \theta$
    LowBound $\leftarrow max($*LowBound*$, \theta + d\theta)$
    for all $r \in F$ **do**
      $[\![e \leq a]\!] \leftarrow$ *Res*$[r]$
      Res$[r] \leftarrow [\![e + u_{F,r} \cdot x_F \leq a]\!]$
  return θ, *Frags*, *Res*, $\tilde{\boldsymbol{x}}$

We then use the LINFBP column generation implementation inside a branch-and-price search. We assume that fragments use redundant resources for the purposes of branching. In particular we rely on each fragment to use 1 unit of a resource that uniquely identifies that fragment, so that we eventually find an integral solution if one exists.

function FBP(*Gs*, δ)
  Fs $\leftarrow \{v_g \mid g \in Gs\}$
  Rs $\leftarrow \{g : [[1 \cdot v_g = 1]] \mid g \in Goals\}$
  LowBound $\leftarrow 0$
  UpBound $\leftarrow M \cdot \|Goals\|$
  Queue $\leftarrow \{\emptyset\}$
  Fs $\leftarrow$ *Fs* $\cup$ *initfrags*(*Gs*, δ)
  while $(1-\epsilon) \cdot$ *UpBound* $>$ *LowBound* **do**
    BranchRs $\leftarrow Pop($*Queue*$)$
    LRs $\leftarrow$ *Rs* $\cup$ *BranchRs*
    θ, *Fs*, *Rs*, $\tilde{\boldsymbol{x}} \leftarrow$ LINFBP(*Fs*, *LRs*, *Gs*, δ)
    if any resources have fractional usage **then**
      if soft constraints satisfied i.e. $\theta < M$ **then**
        $X \leftarrow$ some fractional resource
        Branches $\leftarrow$ branch on $\lceil X \rceil$ and $\lfloor X \rfloor$
        Queue $\leftarrow$ *Queue* $\cup$ *Branches*
    UpBound $\leftarrow$ *SolveIP*(*Fs*, *LRs*)
    LowBound $\leftarrow$ minimum θ in *Queue*

This process can be modified to incrementally return each solution to *SolveIP*(*Fs*, *Rs*) as it is computed, and make this an effective anytime algorithm.

Initial Results

In the bulk freight rail scheduling problem (BFRSP; Described in Davies et al. 2014), we observe speed-ups of several orders of magnitude versus state-of-the-art planners, as seen in Table 1 (Davies et al. 2014).

$\|V\|$	$\|O\|$	m/p	*Golog*	popf2	cpx	FBP
6	2	1	0.4	**0.3**	0.7	1.6
6	4	1	—	—	**2.3**	3.3
6	4	2	—	—	**1.7**	2.0
6	8	2	—	—	**7.5**	7.6
6	16	2	—	—	37.9	**29.7**
6	32	2	—	—	—	**50.3**
6	64	2	—	—	—	**150.3**
6	128	2	—	—	—	**418.8**
6	256	2	—	—	—	**589.7**
16	44	11	—	—	—	**315.0**
16	88	11	—	—	—	**363.0**

Table 1: BFRSP: Time to first solution for increasing track network vertices, $||V||$, and orders , $||O||$, in seconds (1800s time limit, 4GB memory)

References

[Blom and Pearce 2010] Blom, M. L., and Pearce, A. R. 2010. Relaxing regression for a heuristic GOLOG. In *STAIRS 2010: Proceedings of the Fifth Starting AI Researchers' Symposium*, 37–49. Amsterdam, The Netherlands: IOS Press.

[Bonet 2013] Bonet, B. 2013. An admissible heuristic for SAS+ planning obtained from the state equation. In Rossi, F., ed., *International Joint Conference on Artificial Intelligence (IJCAI)*, 2268–2274. IJCAI/AAAI.

[Briel et al. 2007] Briel, M.; Benton, J.; Kambhampati, S.; and Vossen, T. 2007. An LP-based heuristic for optimal planning. In Bessière, C., ed., *Principles and Practice of Constraint Programming - CP 2007*, volume 4741 of *Lecture Notes in Computer Science*. Springer. 651–665.

[Coles et al. 2008] Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008. A hybrid relaxed planning graph-LP heuristic for numeric planning domains. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 08)*, 52–59.

[Davies et al. 2014] Davies, T.; Pearce, A. R.; Stuckey, P. J.; and Søndergaard, H. 2014. Fragment-based planning using column generation. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

[Davies et al. 2015a] Davies, T.; Pearce, A. R.; Stuckey, P. J.; and Lipovetzky, N. 2015a. Sequencing operator counts. In *International Conference on Automated Planning and Scheduling (ICAPS)*. (Accepted).

[Davies et al. 2015b] Davies, T.; Pearce, A. R.; Stuckey, P. J.; and Søndergaard, H. 2015b. Optimisation and relaxation in the situation calculus. In *Autonomous Agents and Multiagent Systems (AAMAS)*. (Accepted).

[Desaulniers, Desrosiers, and Solomon 2005] Desaulniers, G.; Desrosiers, J.; and Solomon, M. M. 2005. *Column Generation*. Springer.

[Fisher 2004] Fisher, M. 2004. Temporal Development Methods for Agent-Based. *Autonomous Agents and Multi-Agent Systems* 10(1):41–66.

[Giacomo, Lesperance, and Levesque 2000] Giacomo, G. D.; Lesperance, Y.; and Levesque, H. J. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121(1-2):109–169.

[Kelly and Pearce 2006] Kelly, R. F., and Pearce, A. R. 2006. Towards high level programming for distributed problem solving. In Ceballos, S., ed., *IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-06)*, 490–497. IEEE Computer Society.

[Lemaréchal 2001] Lemaréchal, C. 2001. Lagrangian relaxation. In Jünger, M., and Naddef, D., eds., *Computational Combinatorial Optimization*, volume 2241 of *LNCS*. Springer. 112–156.

[Mehlhorn and Ziegelmann 2000] Mehlhorn, K., and Ziegelmann, M. 2000. Resource constrained shortest paths. In Paterson, M., ed., *Algorithms—ESA 2000*, volume 1879 of *LNCS*. Springer. 326–337.

[Reiter 2001] Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

[Röger and Nebel 2007] Röger, G., and Nebel, B. 2007. Expressiveness of ADL and Golog: Functions make a difference. In *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, 1051–1056. AAAI Press.