ICAPS 2005
Monterey, California

ICAPS05

POS

Poster Session

Jim Blythe
USC Information Sciences Institute, USA

ICAPS 2005
Monterey, California, USA
June 6-10, 2005

Poster Session

Jim Blythe
USC Information Sciences Institute, USA

Poster Session

Table of contents

ICAPS 2005
Monterey, California

Poster Session

Preface

The ICAPS poster session provides an opportunity to discuss innovative work in progress and late-breaking results. Twenty-two submissions were received, covering a wide range of topics including scheduling, probabilistic planning, BDI systems and autonomic computing. Out of these, ten posters have been selected. I am most grateful to the authors and the reviewers who contributed to this high-quality and very interesting session.

Organizer

- *Jim Blythe*

Programme Committee

- *Chris Beck*
- *Blai Bonet*
- *Amadeo Cesta*
- *Maria Fox*
- *Robert P. Goldman*
- *Patrik Haslum*
- *Joerg Hoffman*
- *Craig Knoblock*
- *Derek Long*
- *Lee McCluskey*
- *David Morley*
- *Marc Spraragen*
- *Biplav Srivastava*

Planning with Numerical State Variables through Mixed Integer Programming

Menkes van den Briel
Department of Industrial Engineering
Arizona State University
Tempe AZ, 85287-8809
menkes@asu.edu

Subbarao Kambhampati
Department of Computer Science
Arizona State University
Tempe AZ, 85287-8809
rao@asu.edu

Thomas Vossen
Leeds School of Business
University of Colorado at Boulder
Boulder CO, 80309-0419
vossen@colorado.edu

Abstract

We extend the state-of-the-art IP formulations for classical planning to include resources and optimization objectives. We present our initial findings and show some preliminary results.

Introduction

One of the most compelling reason for using integer programming (IP) and mixed integer programming (MIP) techniques in planning is when the planning problem contains numerical state variables. Numerical state variables appear in many practical planning domains and are often accompanied by linear numerical constraints and optimization criteria, which are naturally supported by the IP framework. Traditionally, IP has been used to tackle hard combinatorial optimization problems that arise in the field of operation research. However, recent work has shown that IP techniques also show great potential in their ability to solve classical AI planning problems and can compete with the most efficient SAT-based encodings (van den Briel, Vossen, & Kambhampati 2005).

Currently, we are investigating the use of IP techniques in numerical planning by extending the state-of-the-art IP formulations for classical planning and by adding on the work of Kautz and Walser (1999). We will exploit the strength of IP techniques to solve optimization problems with numerical constraints, to extend to AI planning problems that involve numerical state variables and numerical constraints. Even though we are still in the early stages of our research, our first observations and preliminary results suggest that we can improve previous IP approaches to solve these type of planning problems more effetively. Below we will give a brief summary of our IP formulations and show some initial results.

Numerical State Variables

We often refer to numerical state variables as *resources*. Heuristics for planning with resources have been studied by several different works (Do & Kambhampati 2001; Halsum & Geffner 2001; Hoffmann 2002; Refandis & Vlahavas 2000). Studies on IP formulations for resource planning, however, are not as plentiful. Wolfman and Weld (1999) use LP formulations in combination with a satisfiability-based planner to solve resource planning problems, and Kautz and Walser (1999) use IP formulations for resource planning problems that incorporate action costs and complex objectives.

In order to reason about resources, actions are extended to include resource preconditions and effects. Koehler (1998) provides a general framework in which a resource precondition is a simple linear inequality that must hold in each state where the action is applicable, and the action effects are to produce (increase), consume (decrease), or provide (assign) the value of a resource. A resource is called *reusable* if it can only be borrowed, that is, it cannot be consumed or produced by any action. In all other cases a resource is called *consumable*. A reusable resource is *sharable* if it can be borrowed by more than one action at the same time, otherwise it is non-sharable.

The AIPS-2002 planning competition introduced several planning domains with resources. The language that was used in this competition, PDDL2.1 (Fox & Long 2003), incorporates the possibility to define numerical constraints and effects on numerical state variables. Table 1 gives all the numeric domains of this competition and lists all the resource variables. If there exists an action in the domain that has an effect on a resource variable, then that resource variable is listed in the corresponding action effect column. In addition, a type and bounds (where C is some constant) on the resource are given.

We say a resource is *monotonic* if it can only be produced (monotonic$^+$), or if it can only be consumed (monotonic$^-$). We say a resource is *nonmonotonic* if it can both be produced and consumed (nonmonotonic), and if it can be provided (nonmonotonic$^=$). These resource types are used to categorize the resource constraints in our IP formulations.

Integer Programming Formulations

Numerical constraints such as $\sum_{j \in B} a_j x_j + \sum_{j \in C} g_j y_j \leq b$, where a_j and g_j are real numbers, B the set of binary variables, and C the set

Domain	Increase	Decrease	Assign	Type	Bounds
Depots	(current-load ?z)	(current-load ?z)		nonmonotonic	$[0, C]$
	(fuel-cost)			monotonic$^+$	$[0, \infty)$
Driverlog	(driven)			monotonic$^+$	$[0, \infty)$
	(walked)			monotonic$^+$	$[0, \infty)$
Rovers	(energy ?x)	(energy ?x)		nonmonotonic	$[0, \infty)$
	(recharges)			monotonic$^+$	$[0, \infty)$
Satellite	(fuel-used)			monotonic$^+$	$[0, \infty)$
	(data-stored)			monotonic$^+$	$[0, \infty)$
		(fuel ?s)		monotonic$^-$	$[0, C]$
		(data-capacity ?s)		monotonic$^-$	$[0, C]$
Settlers	(available ?r ?v)	(available ?r ?v)	(available ?r ?v)	nonmonotonic$^=$	$[0, \infty)$
	(available ?r ?p)	(available ?r ?p)		nonmonotonic	$[0, \infty)$
	(space-in ?v)	(space-in ?v)		nonmonotonic	$[0, \infty)$
	(labor)			monotonic$^+$	$[0, \infty)$
	(pollution)			monotonic$^+$	$[0, \infty)$
UMT	(weight-load-v ?v)	(weight-load-v ?v)		nonmonotonic	$[0, \infty)$
	(volume-load-v ?v)	(volume-load-v ?v)		nonmonotonic	$[0, \infty)$
	(volume-load-l ?l)	(volume-load-l ?l)		nonmonotonic	$[0, \infty)$
Zenotravel	(onboard ?a)	(onboard ?a)		nonmonotonic	$[0, \infty)$
	(total-fuel-used)			monotonic$^+$	$[0, \infty)$
		(fuel ?a)	(fuel ?a)	nonmonotonic$^=$	$[0, C]$

Table 1: The numeric domains of the AIPS-2002 planning competition

of continuous and integer variables, have received a great deal of attention in the field of mixed integer programming (Savelsbergh 1994). We integrate some of the ideas presented in this field to deal with the numerical constraints and variables that are present in resource planning domains.

Our IP formulations for resource planning are an extension to the IP formulations given by (van den Briel, Vossen, & Kambhampati 2005). In this presentation, we will limit our focus on dealing with the numerical state variables, the propositional variables are dealt with in the same way as in van den Briel, Vossen and Kambhampati (2005). That is, propositional variables are transformed into multi-valued state variables, and changes in the state variables are modeled as flows in an appropriately defined network. As a consequence, the resulting IP formulations can be interpreted as a network flow problem with additional side constraints.

We will use the following notation:

- A: the set of ground actions
- R: the set of resources
- T: the maximum number of plan steps
- $prod(a)$, $cons(a)$, $prov(a)$: the set of resources that appear respectively as produce, consume, provide effects for action $a \in A$
- $produce_{a,r}$, $consume_{a,r}$, $provide_{a,r}$: the amount of resource $r \in R$ that is respectively produced, consumed, provided by action $a \in A$

In our formulations we use actions and numerical state variables, which we define as follows:

- $x_{a,t} \in \{0, 1\}$, for $a \in A, 1 \leq t \leq T$; $x_{a,t}$ is equal to 1 if action a is executed at plan step t, and 0 otherwise.
- $z_{r,t} \geq 0$, for $r \in R, 1 \leq t \leq T$; $z_{r,t}$ represents the value of resource r at plan step t. $z_{r,t}$ can be real or integer-valued and may be bounded from above. For now, we will assume that that each resource has a lower bound that can be normalized to 0.

Numerical state variables add constraints to the planning problem and they may appear in the optimization criteria of the planning problem. Next, we will discuss what constraints need to be added to the IP formulation in order to model the different resources.

Monotonic resources

Resources that behave monotonically can be modeled without introducing numerical state variables to the IP formulation. We can simply deal with these resources by adding them implicitly to the model. Let R^{mon+} and R^{mon-} be the set of resources of type monotonic$^+$ and monotonic$^-$ respectively. If the optimization criteria requires a monotonic resource to be minimized then we can simply setup the following objective function:

$$MIN \sum_{a \in A, 1 \leq t \leq T, r \in R^{mon+} : r \in prod(a)} produce_{a,r} x_{a,t} + \sum_{a \in A, 1 \leq t \leq T, r \in R^{mon-} : r \in cons(a)} consume_{a,r} x_{a,t}$$

Instead of representing monotonic resources by numerical state variables, we can simply deal with them by directly working on the action effects. In case a monotonic resource is bounded then we add an extra constraint to satisfy this bound. For every monotonic$^+$ resource r with an upper bound U_r we must add the following constraint:

$$\sum_{a \in A, 1 \leq t \leq T, r \in R^{mon+}} produce_{a,r} x_{a,t} \leq U_r$$

Similarly, for every monotonic$^-$ resource r with a lower bound L_r and an initial value I_r we must add the following constraint:

$$\sum_{a \in A, 1 \leq t \leq T, r \in R^{mon-}} consume_{a,r} x_{a,t} \leq I_r - L_r$$

Note that in numeric planning domains where all resources are unbounded and monotonic, like the AIPS-2002 numeric driverlog domain, resource planning reduces to classical planning with cost sensitive actions.

Nonmonotonic resources

Resources that are nonmonotonic require the use of numerical state variables in the IP formulation. Since actions may increase or decrease the value of the resource, we need to keep track of their value over time. Let R^{non} be the set of nonmonotonic resources only affected by produce and consume effects of actions, and let $R^{non=}$ be the set of nonmonotonic resources that are affected by provide effects of actions. If nonmonotonic resources are to be minimized then we can add them to the objective function as follows:

$$MIN \sum_{a \in A, 1 \leq t \leq T, r \in R^{non} \cup R^{non=} : r \in prod(a)} produce_{a,r} x_{a,t} + \sum_{a \in A, 1 \leq t \leq T, r \in R^{non} \cup R^{non=} : r \in cons(a)} consume_{a,r} x_{a,t}$$

Also for each nonmonotonic resource $r \in R^{non}$ we must keep track of its value, hence we have the constraint:

$$z_{r,t-1} + \sum_{a \in A, r \in R^{non} : r \in prod(a)} produce_{a,r} x_{a,t} = \sum_{a \in A, r \in R^{non} : r \in cons(a)} consume_{a,r} x_{a,t} + z_{r,t}$$

When an action has a provide effect on a resource $r \in R^{non=}$ then the constraints for keeping track of the resource is more involved. In this case we currently use the linear inequalities as described by Kautz and Walser (1999).

Preliminary Results

For our preliminary studies of the effectiveness of our approaches we compare to the work of Kautz and Walser (KW). We use the airplane domain, which is a modified version of the airplane example from Penberthy and Weld (1994) and Koehler (1998). One or more airplanes can fly between a number of different airports. A fly action consumes fuel and a refuel action provides fuel, hence fuel is a nonmonotonic$^=$ resource. The goal is to take each passenger to his or her destination while minimizing the total fuel consumption. We setup an IP formulation, which we call 1SC, where the propositional variables are modeled as in van den Briel, Vossen, and Kambhampati (2005) and the resource is modeled as in Kautz and Walser (1999). Hence, the main difference in these two approaches is how the propositional variables are modeled. Also, the KW formulation is specifically modeled for this domain with one airplane, whereas our 1SC formulation is domain independent.

		1SC		KW	
Problem	Obj	Nodes	Time	Nodes	Time
(1, 2, 3)	3600	0	0.01	0	0.02
(1, 3, 3)	6780	0	0.02	0	0.04
(1, 6, 3)	9762	15	0.38	56	0.40
(2, 4, 3)	4500	0	0.05		
(2, 5, 3)	5644	42	0.10		
(2, 4, 4)	3939	4	0.14		
(2, 5, 4)	5014	2	0.19		
(2, 6, 4)	9273	606	0.28		
(3, 6, 5)	8914	292	0.49		
(3, 7, 5)	14919	2195	0.71		
(3, 8, 5)	21164	717	0.96		

Table 2: Results for the airplane domain, where the problem number is given by (#airplanes, #passengers, #cities).

Some results for the airplane domain are given in Table 2. All problems were solved to optimality, that is, the objective value represents the minimum amount of fuel needed to transport the passengers. Comparative analysis was made difficult by the fact that the KW formulation only works for single airplane scenarios, while our 1SC formulation can handle multiple airplanes and can be applied to a wide variety of other numerical planning domains. Nevertheless, on the problems that both approaches were able to solve our formulation outperformed the KW formulation in terms of number of branch-and-bound nodes and solution time.

Future Work

Integer programming provides a strong framework for dealing with numerical constraints and optimization criteria. So far, only a few researchers have looked into the application of IP techniques in planning with resources, and we believe that there is significant room for improvement. Even though we are still in the early stages of our research, there are some potential cutting planes that we may be able to detect and add to our IP formulations. For example, the bound constraints on the monotonic resource variables can be interpreted as 0-1 knapsack constraints, and so, we could find knapsack cover inequalities. The constraints on the non-monotonic resources look very similar to constraints we see in lot-sizing problems (Pochet & Wolsey 1995), and so, we could find flow cover inequalities.

Besides adding cutting planes to our IP formulations, we are also looking at different ways to generalize the notion of action parallelism in planning domains that involve resources.

References

Do, M., and Kambhampati, S. 2001. Sapa: A domain-independent heuristic metric temporal planner. In *Proceedings of the European Conference on Planning (ECP-01)*, 109–120.

Fox, M., and Long, D. 2003. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20.

Halsum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *Proceedings of the Sixth European Conference on Planning (ECP-01)*, 121–132.

Hoffmann, J. 2002. Extending FF to numerical state variables. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02*, 571–575.

Kautz, H., and Walser, J. 1999. State-space planning by integer optimization. In *AAAI-99/IAAI-99 Proceedings*, 526–533.

Koehler, J. 1998. Planning under resource constraints. In *Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI-98)*, 489–493.

Penberthy, J., and Weld, D. 1994. Temporal planning with continuous change. In *Proceedings of the 12th National Conference on Artificial Intelligence*, 1010–1015.

Pochet, Y., and Wolsey, L. 1995. *Combinatorial Optimization: Papers from the DIMACS Special Year*, volume 20 of *DIMACS Series in Discrete Mathematics and Computer Science*. American Mathematical Society. chapter Algorithms and reformulations for lot-sizing problems, 245–294.

Refandis, I., and Vlahavas, I. 2000. Heuristic planning with resources. In *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-00)*, 521–525.

Savelsbergh, M. 1994. Preprocessing and probing techniques for mixed integer programming. *ORSA Journal on Computing* 6(4):445–454.

van den Briel, M.; Vossen, T.; and Kambhampati, S. 2005. Reviving integer programming approaches for ai planning: A branch-and-cut framework. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS-05)*, (to appear).

Wolfman, S., and Weld, D. 1999. The LPSAT engine and its applicationto resource planning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 310–317.

Cost Sensitive Conditional Planning

Daniel Bryce & Subbarao Kambhampati
Department of Computer Science and Engineering
Arizona State University, Brickyard Suite 501
699 South Mill Avenue, Tempe, AZ 85281
{dan.bryce, rao}asu.edu

Abstract

While POMDPs provide a general platform for conditional planning under a wide range of quality metrics they have limited scalability. On the other hand, uniform probability conditional planners scale very well, but many lack the ability to optimize plan quality metrics. We present an innovation to planning graph based heuristics that helps uniform probability conditional planners both scale and generate high quality plans when using actions with non uniform costs. We make empirical comparisons with two state of the art planners to show the bene t of our techniques.

Introduction

When agents have uncertainty about their state, they need to formulate conditional plans, which attempt to resolve state-uncertainty with sensing actions. This problem has received attention in both the uncertainty in AI (UAI) and automated planning communities. From the UAI perspective, nding such conditional plans is a special case of nding policies for FOMDPs in the fully observable case, and POMDPs in the partially observable case. The latter is of more practical use, although much harder computationally [Madani *et al.*, 1999; Littman *et al.*, 1998]. The emphasis in the UAI community has been on nding optimal policies under fairly general conditions. However the scalability of the approaches has been very limited. In the planning community, conditional planning has been modelled as search in the space of belief states (which can be seen as a way of characterizing state uncertainty in terms of uniform probability over a set of states). Several planners have been developed–eg. MBP [Bertoli *et al.*, 2001], and PKSPlan [Petrick and Bacchus, 2002] – which model conditional plan construction as an and/or search. These approaches are more scalable than the MDP-based approaches[1], but are often insensitive to the cost/quality information. Indeed, in the presence of actions with differing costs, planners such as MBP (aside from using in-admissible heuristics) can generate plans of arbitrarily low quality, attempting to insert sensing actions without taking their cost into consideration.

In this paper, we describe a way of extending state of the art conditional planners to make them more sensitive to cost/quality information. Our idea is to adapt the type of cost-sensitive reachability heuristics that have proven to be useful in classical and temporal planning [Do and Kambhampati, 2003]. Straightforward adaptation unfortunately proves to be infeasible. This is because in the presence of state uncertainty, we will be forced to generate multiple planning graphs (one for each possible state) and reason about reachability across all those graphs [Bryce and Kambhampati, 2004]. This can get prohibitively expensive–especially for forward searching planners which need to do this analysis once at each search node.

The main contribution of this paper is a way to solve this dilemma. In particular, we propose a novel way of generating reachability information with respect to belief states without computing multiple graphs. Our approach, called the labelled uncertainty graph (LUG) [Bryce *et al.*, 2004], symbolically represents multiple planning graphs, one for each state in our belief, within a single planning graph. Loosely speaking, this single graph unions the support information in explicit multiple graphs and pushes the disjunction, describing sets of possible worlds (states in a belief), into "labels" (ℓ). The graph is built using labels, for sets of worlds, to annotate the planning graph vertices. A label on a vertex signi es the states of our belief that reach the vertex.

To take cost information into account, we describe a method for propagating cost information over the LUG, giving us a cost-propagated LUG ($CLUG$). The (previously mentioned) labels tell us when graph vertices are reachable, but they do not indicate the associated reachability cost. We could track a single cost for the entire set of worlds represented by a label, but this would lose information about differing costs for subsets of the worlds. Tracking a cost for each subset of worlds is also problematic because there are an exponential number of subsets. Instead we track cost over x ed partitions of world sets. This $CLUG$ is used as the basis for doing reachability analysis. In particular, we extract relaxed plans from it (as described in [Bryce *et al.*, 2004]), and use the cost information to help select low cost relaxed plans. Our results show that cost-sensitive heuristics improve plan quality.

We proceed by describing our representation and our planner $POND$, then introduce our planning graph generalization called the $CLUG$, and relaxed plan extraction procedure. We then do an empirical study of the techniques

[1]Their scalability is partly because the complexity is *only* 2-EXP-complete [Rintanen, 2004]!

within our planner and compare with two state of the art conditional planners MBP [Bertoli *et al.*, 2001] and GPT [Bonet and Geffner, 2000]. We end with a conclusion and directions for future work, with emphasis on non-uniform uncertainty.

Representation & Search

$POND$ uses progression search to find strong plans, under the assumption of partial observability. A strong plan guarantees that after a finite number actions executed from any of the many possible initial states, all resulting states will satisfy the goals. The plans are directed acyclic graphs. We define a plan's quality as the expected execution cost (i.e. average path cost), under uniform probability.

$POND$ searches in the space of uniform probability belief states, a technique first described by Bonet and Geffner [2000]. The planning problem P is a tuple $\langle D, BS_I, BS_G\rangle$, where D is a domain, BS_I is the initial belief state, and BS_G is the goal belief state. Belief states are essentially propositional formulas whose models are states. We represent belief states with BDDs. The domain D is a tuple $\langle F, A\rangle$, where F is a set of all fluents and A is a set of actions. Actions have cost and are either causative or observational, with a set of conditional effects or observation formulae as outcomes, respectively.

We use top down AO* search [Nilsson, 1980], in the $POND$ planner to generate conditional plans. In the search graph, the nodes are belief states and the hyper-edges are actions. We need AO* because the application of a sensing action to a belief state in essence partitions the belief state. We use hyper-edges for actions because sensory actions have several outcomes, all if any of which must be included in a solution. The cost model for our plans is the expected execution cost so we use an expectation over the children of a hyper-edge to choose a node's best action.

Cost Labelled Uncertainty Graph ($CLUG$)

To guide search, we use a relaxation of conditional planning to estimate the conditional plan's suffix for each search node. The relaxation measures the cost needed to support the goal by ignoring mutexes between actions, and ignoring sensory actions. While our relaxation does not include sensory actions, the search reasons about the cost of sensing at the current search node. Our heuristic reasons about the transition cost between two sets of states, giving a belief state cost measure[Bryce and Kambhampati, 2004]. The belief state cost measure is a generalization of state cost measures used in classical planning that estimate path costs in the search graph. Following [Bryce and Kambhampati, 2004], we compute the belief state cost measure to estimate the cost of cotransitioning all states in our current belief state to a state in the goal belief state. While we estimate the same metric, we compute it within a single planning graph called the LUG [Bryce *et al.*, 2004] opposed to using a planning graph for every state in our belief state. The LUG was originally developed for unit cost actions, and here we define a generalization of cost propagation techniques for the LUG.

$CLUG$ Construction

We present the $CLUG$, a single planning graph that uses annotations on vertices (actions and literals) to reflect various assumptions about how a vertex (v) is reached. Specifically we use two annotations, a label ($\ell_k(v)$), which denotes the models of our current (source) belief BS_s that reach the vertex at level k and a cost vector reflecting an estimate of the cost of reaching the vertex from different models of the source belief. The annotations help us implicitly represent the vertices common to several of the multiple planning graphs in a single planning graph. The labels for the initial layer literals are used to label the actions and effects they support, which in turn label the literals they support. The use of labels is based on the intuition that (i) actions and effects are applicable in the possible worlds for which their conditions are reachable and (ii) a literal is reachable in all possible worlds where it is given as an effect.

$CLUG$**:** *A levelled graph, where a level k contains three layers, the literal $\mathcal{L}_k$, action $\mathcal{A}_k$, and effect $\mathcal{E}_k$ layers. The construction of the LUG is with respect to the actions in A and a source belief state BS_s. Each LUG vertex is a triple $\langle v, \ell_k(v), c_k(v)\rangle$, where the v is an action a, effect φ_j, or literal l, $\ell_k(v)$ is its label, and $c_k(v)$ is a cost vector.*

Labels: *A label $\ell_k(v)$ of a vertex v is a propositional formula where each model of it is a state $S_s \in \mathcal{M}(BS_s)$. For any such S_s, a classical planning graph built from S_s contains v in level k.*

Without considering cost, as in the LUG, we can use labels to get an idea of the number of worlds in which an action supports a subgoal. A technique we experiment with during relaxed plan extraction is preferring actions that support subgoals in more worlds (coverage) – as we may need to include less actions to support subgoals from all the worlds in BS_s. We do not have to compute cost vectors to use the coverage technique, but it is admittedly myopic because it does not consider the cost of using an action for support. Hence we also experiment with cost vectors for vertices.

Cost Vectors: *A cost vector $c_k(v)$ is a set of pairs $\langle f_i, c_i\rangle$, where f_i is a propositional formula over F and c_i is a rational number. Every c_i is an estimate of the cost of reaching v from all state models $S_s \in \mathcal{M}(f_i)$.*

Cost propagation on planning graphs, similar to that used in the Sapa planner [Do and Kambhampati, 2003], computes the estimated cost of reaching literals at time points. The cached costs give relaxed plans an estimate of the cost associated with including an action (in terms of the cost incurred to support the preconditions of the chosen action). In using the costs, we face a more general scenario where there may be different costs for every subset of models of BS_s. Instead of tracking costs for an exponential number of subsets, we partition the models of BS_s into fixed sets to track cost over (i.e. the elements of the cost vectors $c_k(v)$). The fixed sets are different for every literal, action, and effect, because they are defined with respect to labels. The partitions are with respect to the new worlds that support a vertex at a level. We briefly discuss the procedure for cost and label propagation through graph layers by describing how to find each layer of the graph.

Initial Layer: The LUG has an initial layer, $\mathcal{L}_0$, consisting of every literal that is in a model of BS_s. In the initial layer, the label $\ell_0(l)$ of each literal l represents the states of BS_s in which l holds. In the cost vector, we store a cost of zero for the entire group of worlds in which each literal is initially reachable (i.e. $\langle \ell_0(l), 0\rangle$).

Action Layer: Based on the previous literal layer $\mathcal{L}_k$, the action layer $\mathcal{A}_k$ contains all non-$\perp$ labelled causative actions with from the action set A, plus all literal persistence. The label of the action at level k, is equivalent to the conjunction of the labels of its execution preconditions. If there are new worlds supporting a at level k, we add a formula-cost pair to the cost vector with the formula equal to $\ell_k(a) \wedge \neg\ell_{k-1}(a)$.[2]

We then update the cost for each element of the cost vector, where each f_i is a formula describing the new worlds of BS_s that came to support a at a distinct level. We nd c_i by summing the costs of the execution precondition literals, in the worlds described by f_i. The cost of each literal is determined by covering f_i with the cost vector of the literal.

For example, we wish to compute the cost of an action a in a set of worlds described by f, where a has a single execution precondition l with a cost vector $\{\langle f', 3\rangle, \langle f'', 5\rangle\}$ Say f represents two states s_1, s_2, f' represents s_1, s_3, and f'' represents s_2, s_4. We need both f' and f'' to cover f because both cover one state of f. The cost of f is then 8 because the cost of the cover is 8.

Effect Layer: An effect φ_j is included in $\mathcal{E}_k$, when it is reachable in some world of BS_s, i.e. $\ell_k(\varphi_j) \neq\perp$, which only happens when both the associated action and the antecedent are reachable in at least one world. The cost c_i of world set f_i of an effect at level k is found by adding the execution cost of the associated action, the support cost of the action in the worlds of f_i, and the sum of the cost of the antecedent literals in f_i.

Literal Layer: The literal layer, $\mathcal{L}_k$, contains all literals with non-$\perp$ labels. The label of a literal, $\ell_k(l)$, depends on $\mathcal{E}_{k-1}$ and is the disjunction of the labels of each effect that gives the literal. The cost c_i in a set of worlds f_i for a literal at level k is found by covering the worlds f_i with the union of all formula-cost pairs of effects that support the literal.

Termination: The $CLUG$ construction stops when two consecutive literal layers are identical.

Relaxed Plans

The LUG and $CLUG$ relaxed plan heuristics account for positive world interaction and independence across source states in achieving the goals. In the relaxed plan we support the goal with every state in BS_s, but in doing so we track which states in BS_s use which paths in the graph. There may be several paths used to support a subgoal in the worlds of BS_s, because not one supports all worlds. One challenge in extracting the relaxed plan is in doing the proper label algebra to track what worlds use which paths to support subgoals. Another challenge is in extracting cost-sensitive relaxed plans from the $CLUG$. In the next section, we demonstrate both effectiveness of using cost vectors (cost) as in the $CLUG$ and the size of action's label (coverage) as in the LUG to decide which actions to use to support subgoals.

The LUG and $CLUG$ relaxed plans are inadmissible, i.e. will not guarantee optimal plans with AO* search. Admissible heuristics are lower bounds that enable search to nd optimal solutions, but most in practice are very ineffective for improving search ef cienc y. In the next section we demonstrate that although our heuristics are inadmissible they guide our planner toward solutions of comparable quality to a planner that uses admissible heuristics and do so much faster.

[2]When $k = 0$ we can say $\ell_{-1}(a) =\perp$.

Empirical Comparisons

Our main intent is to evaluate the effectiveness of the LUG and $CLUG$ relaxed plans in improving the quality of plans generated by $POND$. Additionally, we also compare with two state of the art planners, GPT [Bonet and Geffner, 2000], and MBP [Bertoli *et al.*, 2001]. Even though MBP does not plan with costs, we show the expected cost of MBP's plans for each problem's cost model. GPT uses admissible heuristics based on relaxing the problem to full-observability (whereas our relaxation is to no observability while ignoring action mutexes), and MBP uses a belief state's size as its heuristic merit. For lack of space, our test set up involves a single domain, Medical-Specialist. Each problem had a time out of 20 minutes and a memory limit of 1GB on a 2.8GHz P4 Linux machine.

Medical-Specialist: We developed an extension of the medical domain [Weld *et al.*, 1998], where in addition to staining, counting of white blood cells, and medicating, one can go to a specialist for medication and there is no chance of dying – effectively allowing conformant (non-sensing) plans where the specialist medication is used for every disease. We assigned costs as follows: c(stain) = 5, c(count_white_cells) = 10, c(inspect_stain) = X, c(analyze_white_cell_count) = X, c(medicate) = 5, and c(specialist_medicate) = 10. We generated ten problems, each with the respective number of diseases (1-10), in two sets where X = {15, 25}.

Our results in the rst two columns in Figures 1, 2, and 3 show the expected cost, plan breadth, and total time for two cost models. Relaxed plans based on propagated cost instead of coverage enable $POND$ to be more cost-sensitive. Using the cost propagation method, plans tend to branch less than coverage as the cost of sensing increases in order to reduce expected cost. Since MBP is insensitive to cost, the its plans are proportionately costlier as the sensor cost increases. GPT returns better plans, but tends to take significantly more time as the cost of sensing increases; this can be attributed to how the heuristic is computed by relaxing the problem to full-observability. Our heuristics measure the cost of co-achieving the goal from a set of states, whereas GPT takes the max cost of the states.

In summary, the experiments show that the LUG heuristics help with scalability and using propagated cost to extract relaxed plans helps nd better solutions. We also found that planners not reasoning about action cost can return arbitrarily poor solutions, and planners that use weaker assumptions about uncertainty and cost do not scale as well.

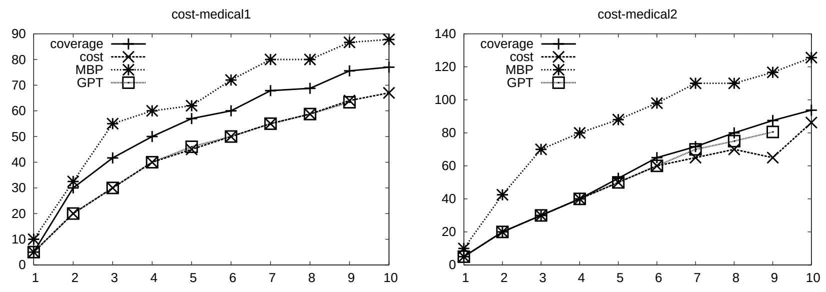


Figure 1: *Expected cost results for $POND$ (coverage and cost), MBP, and GPT for Medical-Specialist.*

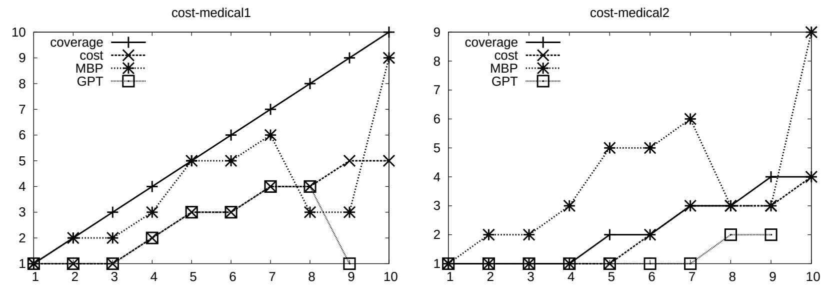


Figure 2: *Breadth (# of plan paths) results for $POND$ (coverage and cost), MBP, and GPT for Medical-Specialist.*

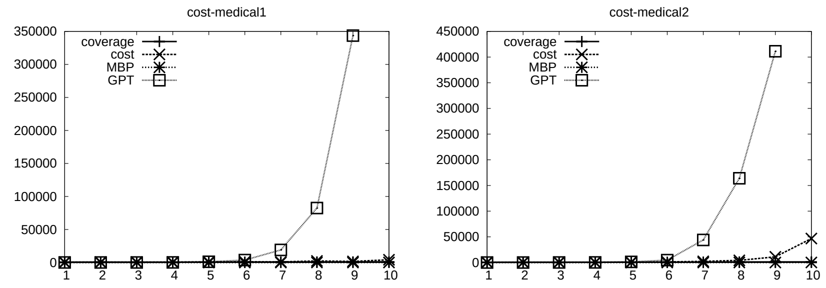


Figure 3: *Total Time(ms) results for $POND$ (coverage and cost), MBP, and GPT for Medical-Specialist.*

Conclusion

With our motivation toward conditional planning approaches that can scale like classical planners, but still reason with quality metrics, we have presented a planning graph innovation called the $CLUG$. With this we extract cost-sensitive relaxed plans that are effective in guiding our planner $POND$ toward high-quality conditional plans. We have shown with an empirical comparison that our approach improves the quality of conditional plans over conditional planners that do not take cost information into account, and we can out scale previous approaches that consider cost information in a weaker fashion. Given our ability to propagate numeric information on the LUG, we are currently adapting these heuristics to handle non uniform probabilities. The extension involves using expected cost in the cost vectors so relaxed plans can select actions such that subgoals are supported with high probability, but low expected cost.

References

P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in non-deterministic domains under partial observability via symbolic model checking. In *Proceedings of IJCAI'01*, 2001.

B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *Proceedings of AIPS'02*, 2000.

D. Bryce and S. Kambhampati. Heuristic guidance measures for conformant planning. In *Proceedings of ICAPS'04*, June 2004.

D. Bryce, S. Kambhampati, and D.E. Smith. Planning in belief space with a labelled uncertainty graph. Technical report, AAAI Workshop TR WS-04-08, 2004.

M. Do and S. Kambhampati. Sapa: A scalable multi-objective heuristic metric temporal planner *JAIR*, 2003.

M. Littman, J. Goldsmith, and M. Mundhenk. The computational complexity of probabilistic planning. *JAIR*, 9:1–36, 1998.

O. Madani, S. Hanks, and A. Condon. On the undecidability of probabilistic planning and in nite-horizon partially observable markov decision problems. In *AAAI/IAAI*, pages 541–548, 1999.

N. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1980.

R. Petrick and F. Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of AIPS'02*, 2002.

J. Rintanen. Complexity of planning with partial observability. In *Proceedings of ICAPS'04*, 2004.

D. Weld, C. Anderson, and D.E. Smith. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of AAAI'98*, 1998.

Replanning: a New Perspective

William Cushing and **Subbarao Kambhampati**
Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406
{wcushing|rao}@asu.edu

Abstract

Replanning involves generating a new plan to fix execution failures. Past work has often characterized replanning procedurally as a special case of Plan Reuse: reuse the current plan to solve a new problem with altered initial and goal states. Just as in normal reuse, these replanning systems focus on minimally perturbing the plan while accounting for the alterations in the problem. There are two important limitations to this view of replanning. First, many failures cannot be represented as alterations to initial and goal states. Second, plan reuse is motivated by efficiency considerations during plan generation, while replanning attempts to minimize the residual *execution* costs of the partial execution.

In this paper, we argue that replanning should rightly be seen as solving a new planning problem that not only captures *general execution failures*, but also the *commitments* incurred during the partial execution. Failures are modeled as fine-grained modifications to operator descriptions, and commitments are modeled as soft constraints. From this general perspective, minimal perturbation planning can be understood as a crude heuristic for respecting commitments: if we keep the plan the same, then we are likely to respect the commitments as well.

Introduction

Replanning techniques resolve execution failures of prior plans. The literature has conflated replanning and plan reuse by viewing replanning as a special case of plan reuse: reuse the current plan on altered initial and goal states. This line of research presents *Minimal Perturbation Planning* as the panacea for both plan reuse and replanning(Kambhampati 1990; Hammond 1986; Simmons 1988). To start with, it is overly optimistic to assume that an execution failure can be represented by a new planning problem with altered initial and goal states. This boils down to assuming that execution failures are independent of the agent's behavior. When this is not the case, making the assumption can lead to indefinitely repeating the failure. To avoid this, we say that a *correct* replanning solution avoids repeating execution failures.

Furthermore, preserving plan structure has little connection to resolving failures. To the contrary, execution failures imply that at least some part of the plan needs to be altered. Trying to preserve structure is, therefore, a handicap. Yet, there is a connection: minimal perturbation planning can be seen as a heuristic approach to keeping overall execution costs low when replanning in the presence of collaborators. In the presence of collaborators, altering one's intentions can degrade overall execution performance: external agents could have based their own plans off of one's stated intentions. We refer to such a dependency as a *commitment*, and say that the *quality* of a replanning solution is in terms of respecting commitments.

A Motivating Scenario: In this paper, we take the perspective that replanning is the projection of a multi-agent planning and execution problem onto a single-agent. Our motivating example is $\mathrm{AltAlt}^{\mathrm{PDA}}$, a hypothetical personal digital assistant equipped with an automated planner. Its equally hypothetical user, Romeo, uses $\mathrm{AltAlt}^{\mathrm{PDA}}$ to automatically produce efficient travel plans. We will consider equipping $\mathrm{AltAlt}^{\mathrm{PDA}}$ with a replanning capability in order to support the dynamic execution of these travel plans by Romeo.

The full problem Romeo faces is very complex, as his travels take him to distant and unfamiliar places. This involves interacting with many other agents, directly and indirectly. Formalizing this situation as a full multi-agent problem would allow $\mathrm{AltAlt}^{\mathrm{PDA}}$ to produce globally optimal plans. Of course, Romeo's colleagues may not be very impressed with the endless negotiations carried out on Romeo's behalf. We presume that even though there are other agents, it is unnecessary to take them into consideration in order to synthesize efficient travel plans. However, once a plan has been published, it is no longer possible to completely ignore the external agents: the commitments induced by the published plan must be considered. Therefore, we project the multi-agent problem onto a single-agent problem by ignoring everything except the static impact of the external agents: the projection abstracts every aspect of the external agents except the current set of commitments.

Contribution: In this paper, we consider how to update the original planning problem so that solutions resolve failures while respecting commitments. Doing so involves two distinct modeling issues: a language for failure representation, and a language for commitment representation. We present a particular language for representing failures, and define the correctness of a replanning solution with respect to it. From there, we consider the somewhat orthogonal issue of representing commitments. We model a commitment as a soft constraint, and thereby reduce replanning solution quality to partial-satisfaction planning solution quality. Starting from this high level discussion, we present a formal model of replanning with failures and commitments in terms of metric-temporal partial-satisfaction planning. Before concluding, we discuss some of the connections to related work in plan reuse and real-world systems.

Correctness: Execution Failure

The ultimate goal in replanning is to produce a new executable, relevant, failure-resolving plan. Unfortunately, traditional planning theories only verify two of these properties: executability from an initial state, and relevancy to a goal description. To ensure that solutions further resolve failures, some extension of planning theory is required. Here, we are motivated by the concept of replanning as "planning again". That is, while some researchers (Hammond 1986; Simmons 1988) consider augmenting traditional planning theories with techniques for directly resolving failures, we instead consider reducing the replanning problem to a modified planning problem.

These modifications to the planning problem are to ensure that executability and relevancy with respect to the modified instance, together, implies executability, relevancy, and failure-resolution with respect to the executor's updated theory. This is trivial when the executor has no ability to learn from its mistakes; unexpected dynamics are simply represented as changes to the initial state and to the goal description. Given that we have assumed an environment where failures occasionally occur, executors incapable of learning are doomed for all but the most restricted case: those environments where failures are entirely independent of the behaivor of the agent.

For example, let's suppose that Romeo has used $\mathrm{AltAlt}^{\mathrm{PDA}}$ to synthesize a travel plan to Los Angeles, specifically, one which involves driving. Suppose Romeo attempts to start the car, and that the attempt fails. If we restrict ourselves to representing failures as changes in the initial state and the goal description, then Romeo has a dilemma: the initial state is unchanged, as is the goal. The old plan would still be valid! It is nonetheless clear that the old plan is inappropriate; specifically, Romeo would possess the knowledge that future attempts to start the car will continue to fail.

For these reasons, we describe a failure as a fine-grained modification to the planning instance. The semantics of traditional planning theories are typically given in terms equivalent to Markov Decision Processes, where the planner chooses among several transition matrices at each state. In the case of deterministic planning, for example, such matrices consist of only 0's and 1's, whereas in probabilistic planning, rows encode the posterior distribution of states given the execution of an action. For our purposes, we say a failure can be described as a set of modifications to such matrices. As an example, the transition matrix for driving, in the replanning scenario described above, would be altered by the failure to the identity matrix: every attempt at driving will fail to achieve a change in state. After applying such a set of modifications, the definition of executability further entails failure-resolution: so we have reduced replanning correctness to planning correctness.

Quality: Commitments

In many, if not all, potential applications of automated planning, plan quality is just as important as plan correctness. We can assume that an appropriate quality metric has been provided for the original planning domain, but after an execution failure, it is not clear if that metric is still appropriate. For example, many authors claim that a more appropriate quality metric is preservation of prior plan structure. Some high-level justifications for ignoring the old quality metric in favor of prior plan syntax include rescheduling cost, commitment preservation, and user acceptance in mixed-initiative planning. In fact, these three explanations are really but different facets of the presence of external, collaborative, unmodeled agents.

This raises some important questions:

1. Why did the original model omit the external agents?
2. How does plan failure invalidate that reason?
3. What is the right model of this expanded quality metric?

Full multi-agent planning and execution is very complex. It is difficult to formalize, acquire models, and find solutions. Moreover, in many settings, the presence of other agents ends up being irrelevant, as an efficient plan exists to achieve the goal independently of the external agents. For these reasons, it is common to project away all external agents, including friends, in order to simplify a complex multi-agent problem into a single-agent planning problem. In our running example, it is clearly the case that Romeo, or his assistant, can find efficient travel plans without consulting potential collaborators.

Before any decisions have been reached, there are no dependencies between agents at all. A friend can hardly be justified in complaining if one fails to rendezvous, when no rendezvous was arranged. Sometime after the a plan has been constructed, however, it is possible that communication of the plan took place. If so, then the agent might now be committed to some of those decisions. For example, suppose we consider a scenario where Romeo is traveling to Los Angeles to attend a conference. Sometime during the execution, he learns that the conference has been canceled, as the hotel has burned down. If, after making the travel plan, Romeo ended up making arrangements to meet with his friend Bob, who dwells in Los Angeles, then even after the hotel has burned down Romeo does still have a commitment to meet Bob at the prearranged place in Los Angeles.

Suppose we try to capture this altered notion of plan quality with minimal perturbation planning; the new quality metric rewards plans which preserve old structure. In the scenario where the hotel burns down, preserving old plan structure keeps all the traveling actions, thus allowing Romeo to meet with Bob in Los Angeles: preserving the commitment. Of course, even if there had been no commitment, the traveling would still have been performed – without any benefit to Romeo at all. Moreover, the cost of driving and arranging new lodging may far outweigh the cost of canceling the meeting with Bob. Clearly, arbitrarily preserving syntax is not adequately capturing a measure of plan quality.

Instead, we say that a commitment introduces a new goal in the altered problem instance. Typically, this would be a *soft constraint*, as failing to meet a commitment is not often grounds for completely giving up on any course of action. So, in fact, we are considering representing replanning as alterations to an underlying partial-satisfaction planning problem: where the soft constraints are commitments.

These commitments arise after $\mathrm{AltAlt}^{\mathrm{PDA}}$ automatically synthesizes a plan for Romeo. At that point, Romeo might communicate some aspects of this plan to the many external agents he interacts with. These agents can then request that Romeo *commit* to some or all of those communicated aspects. Given that Romeo agrees to do so, then he would be able to model this commitment as a new goal for $\mathrm{AltAlt}^{\mathrm{PDA}}$ to achieve. Of course, the current plan already achieves any such commitment, so there isn't any need to alter the plan in response to the new goal until after a failure occurs. However, after a failure occurs, the goals introduced by external commitments serve as an adequate measure of plan quality; failing to achieve such a goal fails to maintain the associated commitment (and so fails to accrue the associated reward).

Formal Treatment

To fully ground our presentation of failures and commitments we demonstrate altering instances of metric-temporal partial-satisfaction (van den Briel *e*t al. 2004) planning problems to account for execution failures and commitments to external agents. We use metric time to properly motivate the interactions between collaborating agents; weaker forms of planning do not admit a description of some method to synchronize the activities of agents. We presume that an appropriate interface exists (such as a parser, or a GUI) between the top-level user and the automated replanner, specifically, we will assume that the input to the replanner representing failures and commitments has already been formalized in the following manner:

Definition 1 (Failure) *A failure, f, is a 4-tuple, (S, A, P, E), of strings in PDDL syntax. S and P are goal descriptions, A is an operator name followed by a (parenthesized) list of variables and constants, and E is an effect. A failure is activated when the agent attempts to execute some binding of A in some state satisfying S; the activation alters the preconditions and effects of that action application. Specifically, the action is only executable if it further satisfies P, and the resultant state is obtained by additionally applying the effect E. Conflicts between E and the normal effects of the action are resolved in favor of E: E is applied after the normal effects of the action. However, antecedents of conditional effects in E are still evaluated with respect to the original state, despite the fact that E is applied to an intermediate state.*

For example, suppose Romeo notices that *drive(carA,PHX,LA)* has failed, and further surmises that the fault is due to *carA* being broken. This failure could be represented as: *(true,drive(carA,?x,?y),true,no-op)*. That is, in every state of the domain, attempting to drive using *carA* will activate the failure. The failure manifests itself by causing the drive action to be ineffectual; *no-op* is an abbreviation for the set of conditional effects of the form *(when p p)* for every literal p. That is, the resultant state must be identical to the original state, so that attempting to drive with *carA* becomes a no-op.

Definition 2 (Commitment) *A commitment to an external agent, $c = (G, r)$, is a 2-tuple consisting of a goal description and an associated reward (which can be a special symbol infinity for a hard goal).*

In the context of metric-temporal planning, we assume goals are given as achievement formula within a single window of opportunity, where the right side, the deadline, is typically the more challenging constraint to satisfy. Goal-achievement of a plan is then defined as achieving each formula at some point within each associated window. The reward accrued by achieving soft goals is simply the sum of the associated rewards, and the utility of a plan is its aggregate reward minus its aggregate cost; aggregate cost is simply the sum of the costs of each action.

From here, we can modify the ground instance of a partial-satisfaction metric-temporal planning problem by adding the commitments to the list of goals and altering the ground operators according to the failure descriptions. The last step sometimes involves splitting ground operators, as a ground operator actually corresponds to many transitions in the graph; a failure can theoretically modify just a single transition.

Example: Suppose Romeo's summer plans not only included the conference in Los Angeles, but also a short educational visit to Boston a month later. Further suppose that Romeo has stipulated a large number of additional places to visit in both Los Angeles and Boston, so that his assistant, $\mathrm{AltAlt}^{\mathrm{PDA}}$, has generated a custom walking tour of both cities, in addition to the rest of the travel plan. Romeo has told Bob of his plans, and they mutually agreed to meet at the California Science Center at noon. Fate is kinder than in the preceding sections: the only failure encountered is that Romeo discovers that he has forgotten to pack his sneakers. As he is hardly willing to engage in the walking tour of Los Angeles in dress shoes, he decides to use $\mathrm{AltAlt}^{\mathrm{PDA}}$ to replan. So, in this case, the tour should be regenerated, favoring public transportation over walking, and preferring to keep the noon appointment with Bob. In uncomfortable shoes, suppose Romeo is willing to walk half as far (500 m.) to a specific destination, and only up to a quarter (5 km.) of the normal maximum distance in any given day. Since Romeo expects to remember to pack comfortable shoes for the trip to Boston, he limits this failure to his current stay in Los Angeles.

The formal representation of the failure is given by: *(at(Los Angeles), walk(?s,?d), (and (< distance(?s,?d) 500) (< walked_today (- 2500 distance(?s, ?d)))), nil)*. Likewise, the commitment to meet Bob is just *($\langle$at(California Science Center), noon - 5 min, noon + 5 min$\rangle$, r)*, where r is presumably large, since canceling the day of an appointment with the weak excuse of not having comfortable shoes would probably offend Bob, let alone inconvenience him.

The ground planning instance would consist of all the old goals, along with the commitment to Bob, as well as a split version of the walk operator. The original walk operator $\mathrm{walk}_{\mathrm{normal}}$, would have an additional precondition of being applicable only outside of Los Angeles. The walking in uncomfortable shoes operator, $\mathrm{walk}_{\mathrm{fail}}$, would be applicable in Los Angeles, but would additionally require that the destination be half as far, and that the total distance traveled would be less than a quarter of its usual maximum.

Synthesizing an entirely new plan with respect to this planning problem resolves the failure; the Los-Angeles specific goals would be accomplished by alternative means, where necessary, including the explicit commitment to Bob. After leaving Los Angeles the planning instance behaves the same as when $\mathrm{AltAlt}^{\mathrm{PDA}}$ generated the original plan for Romeo. Therefore, the Boston portion of the plan would remain unaffected, just as in minimal perturbation planning. However, minimal perturbation planning would often miss the commitment to Bob. Suppose Romeo is not visiting any attractions within 500 meters of the California Science Center. Keeping the commitment requires perturbing the prior plan: walking to, from, or past the science center cannot be reused.

Related Work

Proponents of replanning as plan reuse, as exemplified by (van der Krogt & de Weerdt 2005), have doubly confused the issue of replanning quality. The first approximation, considered at length in this paper, is in assuming that reducing perturbation caused to other agents can be modeled by minimally altering the structure of plans across iterations: minimal perturbation planning. The second approximation is applying their plan reuse algorithm instead of truly minimal perturbation planning. That is, minimal perturbation planning is just as inadequate in application to plan reuse as it is in application to replanning – though for different reasons (Nebel & Koehler 1995). Specifically, minimal perturbation planning has greater complexity than plan synthesis, which is in direct conflict with the speedup motivation of plan reuse. As noted by Nebel & Koehler, plan reuse systems actually return highly, but not maximally, similar plans.

The robot path planning community (Stentz 1995; Koenig, Likhachev, & Furcy 2004) has long looked at replanning slightly differently from the planning community. In particular they try to ensure that the plan produced by the ”re-planner” is as optimal as the one that would have been produced by the from-scratch planner. Due to the nature of robot path planning, this work does not consider the commitments made by the partial execution of the prior plan. One point of similarity is that the robot path planning community does model failures that involve more than initial and goal state changes–action deletion (e.g. certain navigation actions made infeasible by the apperance of new obstacles). This is a kind of systematic failure, which we generalize further based on accounts of real-world replanning systems(Pell *e*t al. 1997; Myers 1999).

Within the planning community, the work by Pell *e*t al. comes closest to recognizing the importance of respecting the commitments generated by a partially executed plan. They however do not give any formal details about how the replanning is realized in their system.

Conclusion

The observation underlying our work is that replanning lacks a formal definition. At a high level, researchers bring up the concepts of commitments, reservations, approximate domain models, mixed-initiative and distributed planning, tightly bounded computational resources, and execution failures. Then a syntactic measure is introduced without any formal connection to these motivations. While approximating replanning using minimal perturbation planning could work well empirically, demonstrating this first requires an idea of non-approximate replanning.

We tackle this problem by considering the intuitively optimal behavior for a set of interesting replanning scenarios. We find that it is easy to construct scenarios where optimal behavior can be explained in terms of systematic execution failures and commitments to external agents. We develop a formal model of replanning that directly captures systematic failures and commitments, and admits as formal solutions only those that properly respond to the additional constraints. Given that those are transcribed correctly, then the formal solution of the model corresponds to the intuitively optimal behavior first described. Our model thus serves as a formal definition of the kind of replanning concerned with resolving execution failures while respecting commitments.

References

Hammond, K. J. 1986. Chef: A model of case-based planning. In *A*AAI, 267–271.

Kambhampati, S. 1990. Mapping and retrieval during plan reuse: A validation structure based approach. In *A*AAI, 170–175.

Koenig, S.; Likhachev, M.; and Furcy, D. 2004. Lifelong planning a. *A*rtif. Intell. 155(1-2):93–146.

Myers, K. L. 1999. Cpef: A continuous planning and execution framework. *A*I Magazine 20(4):63–69.

Nebel, B., and Koehler, J. 1995. Plan reuse versus plan generation: A theoretical and empirical analysis. *A*rtif. Intell. 76(1-2):427–454.

Pell, B.; Gat, E.; Keesing, R.; Muscettola, N.; and Smith, B. 1997. Robust periodic planning and execution for autonomous spacecraft. In *I*JCAI, 1234–1239.

Simmons, R. G. 1988. A theory of debugging plans and interpretations. In *A*AAI, 94–99.

Stentz, A. 1995. The focussed d* algorithm for real-time replanning. In *I*JCAI, 1652–1659.

van den Briel, M.; Nigenda, R. S.; Do, M. B.; and Kambhampati, S. 2004. Effective approaches for partial satisfaction (over-subscription) planning. In *A*AAI, 562–569.

van der Krogt, R., and de Weerdt, M. 2005. Plan repair as an extension of planning. In *I*CAPS.

Incremental Hashing for Pattern Databases

Stefan Edelkamp and Tilman Mehler
Computer Science Department
University Dortmund
{stefan.edelkamp,tilman.mehler}@cs.uni-dortmund.de

Abstract

Hashing is essential for state-space search to avoid redundant work and to address abstract states in pattern databases. But in some domains hashing becomes a bottleneck of the exploration. This applies in particular to planning with pattern databases where both the actual and the abstract state need to be hashed, each requiring time linear in the number of atomic propositions. In this paper, we devise an incremental hashing scheme for planning with pattern databases, which reduces the time complexities for hashing to a constant. We exemplify our considerations in two established planning domains.

Introduction

The effectiveness of *Iterative Deepening A** (IDA*) (Korf 1985) for solving problems with heuristic search, also applies to action planning. As backtracking keeps the changes in the state representation vector during the exploration small, the algorithm is often tuned to maximize the number of nodes per second. Since action planning belongs to the class of problems, whose exploration yields a lot of duplicate states, hashing is essential. Many pruning techniques for IDA* typically rely on the regular structure of the state space graph (Taylor & Korf 1993). On the other hand, IDA* with static transposition tables (Reinefeld & Marsland 1994) frequently first exhausts memory, then time. One compromise of memory-sensitive A* and time-sensitive IDA* search is known as *Partial IDA** (Hüffner *et al.* 2001), which leads to much larger hash table sizes and to a better state space coverage. Even though completeness is sacrificed, the expected error probability is small.

The time to compute the hash address can result in a bottleneck for the exploration. Even if all other operations to generate a successor for a given state are of constant time, a non-incremental computation of a hash function for a state vector of size k accumulates to $\Omega(k)$ time. The larger the state vector, the better the savings that can be obtained by incremental state space hashing. For partial search, incremental hashing leads to $O(1)$ time per state lookup. The paper is structured as follows. We first review the algorithm of Rabin and Karp that was designed to accelerate pattern matching. We then discuss, how this method can be extended to state space search and exemplify this consideration on STRIPS planning. Next, we address incremental and perfect hashing for pattern database construction and usage. The proposed algorithm will apply incremental hashing for the original and abstract state spaces. We implement this technique to domain-independent STRIPS action planning and perform experiments in two established planning domains.

Rabin and Karp's Algorithm

The idea of incremental hashing originates in matching text $T[1..n]$ to a pattern $M[1..m]$. In the algorithm of (Karp & Rabin 1987), a pattern M is mapped to a number $h(M)$, which fits into a single memory cell and can be processed in constant time. For $1 \leq j \leq n - m + 1$, it will check if $h(M)$ is equal to $h(T[j..j + m - 1])$. Due to possible collisions, this is not a sufficient but a necessary criterion for the match of M and $T[j..j + m - 1]$. A character-by-character comparison (check) is needed only if $h(M)$ equals $h(T[j..j + m - 1])$.

In order to compute $h(T[j + 1..j + m])$ incrementally in constant time, we take value $h(T[j..j+m-1])$ into account, according to Horner's rule for evaluating polynomials. Let $q > m$ be a sufficiently large prime. We assume that numbers of size $q \cdot |\Sigma|$ fit into a memory cell, so that all operations can be performed with single precision arithmetic. To ease notation, we identify characters in Σ with their order. The algorithm of Rabin and Karp performs the matching process. The input is a string T and a pattern M. The output is the first occurrence of M in T, if any. The stages of the algorithm are:

1. Initialize $p \leftarrow t \leftarrow 0$ and $u \leftarrow |\Sigma|^{m-1}$ mod q
2. Precompute hash value of the pattern, i.e., for all $1 \leq i \leq m$ set $p \leftarrow (|\Sigma| \cdot p + M[i])$ mod q
3. Precompute hash value of the text prefix, i.e., for all $1 \leq i \leq m$ set $t \leftarrow (|\Sigma| \cdot p + T[i])$ mod q
4. Iterate for all $1 \leq j \leq n - m + 1$:
   (a) if hash function values match, e.g. if ($p = t$), then call procedure check $(M, T[j..j + m - 1])$
   (b) if pattern is found then return j
   (c) if $j \leq n - m$, use Horner's rule to compute $t \leftarrow ((t - T[j] \cdot u) \cdot |\Sigma| + T[j + m])$ mod q

As an example take the alphabet $\Sigma = \{0, \ldots, 9\}$ and $q = 13$. Furthermore, let $M = 31,415$ and $T = 2,359,023,141,526,739,921$. The mapping induced by function h yields the following sequence of hash values: $8, 9, 3, 11, 0, 1, 7, 8, 4, 5, 10, 11, 7, 9$, and 11. We see h produces collisions. The incremental computation at $j = 8$ works as follows: $14,152 \equiv (31,415 - 3 \cdot 10,000) \cdot 10 + 2 \pmod{13} \equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13} \equiv 8 \pmod{13}$. The computation of all hash addresses has a running time of $O(n + m)$; the best case for the matching process.

Incremental Hashing

For state space search, we often have the case that a state transition changes only a small part of the state vector representation. In this case, the computation of the hash function can be designed incrementally. The difference to the algorithm above is that changes can now occur at intermediate index positions within the state vector.

A *propositional planning problem* (in STRIPS notation) is a finite state space problem $\mathcal{P} =< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$, where $\mathcal{S} \subseteq 2^{AP}$ is the set of states, $\mathcal{I} \in \mathcal{S}$ is the initial state, $\mathcal{G} \subseteq \mathcal{S}$ is the set of goal states, and $\mathcal{O}$ is the set of operators that transform states into states. Operators $o = (P, A, D) \in \mathcal{O}$ have propositional preconditions P, and propositional effects (A, D), where $P \subseteq AP$ is the *precondition list*, $A \subseteq AP$ is the *add list* and $D \subseteq AP$ is the *delete list*. Given a state S with $P \subseteq S$ then its successor $S' = o(S)$ is defined as $S' = (S \setminus D) \cup A$.

It is not difficult to devise an incremental hashing scheme for STRIPS planning that bases on the idea of the algorithm of *Rabin and Karp*. For $S \subseteq AP$ we may start with $h(S) = (\sum_{p_i \in S} 2^i) \bmod q$. The hash value of $S' = (S \setminus D) \cup A$ is

$$\begin{aligned} h(S') &= \left(\sum_{p_i \in (S \setminus D) \cup A} 2^i \right) \bmod q \\ &= \left(h(S) - \sum_{p_i \in D} 2^i + \sum_{p_i \in A} 2^i \right) \bmod q. \end{aligned}$$

Since $2^i \bmod q$ can be pre-computed for all $p_i \in AP$, we have a running time that is of order $O(|A| + |D|)$, which is constant for most STRIPS planning problems. It is also possible to achieve constant time complexity if we store the values $\mathit{inc}(o) = (\sum_{p_i \in A} 2^i) \bmod q - (\sum_{p_i \in D} 2^i) \bmod q$ together with each operator. Either complexity is small, when compared to ordinary hashing of the planning state.

Pattern Database Search

Abstraction functions ϕ map states $S = (S_1, \ldots, S_k)$ to patterns $\phi(S) = (\phi(S_1), \ldots, \phi(S_k))$. Pattern databases (Culberson & Schaeffer 1998) are hash tables for fully explored abstract state spaces, storing with each abstract state the shortest path distance in the abstract space to the abstract goal. They are constructed in a complete traversal of the inverse abstract search space graph. Each distance value stored in the hash table is a lower bound on the solution cost in original space and serves as a heuristic estimate. Different pattern databases can be combined either by adding or maximizing the individual entries for a state.

Pattern databases work, if the abstraction function is a homomorphism, so that each path in the original state space has a corresponding one in the abstract state space. In difference to the search in original space, the entire abstract space has to be looked at. As pattern databases are themselves hash tables we apply incremental hashing, too.

If we restrict the exploration in STRIPS planning to some certain subset of propositions $R \subseteq AP$, we generate a planning state space homomorphism ϕ and an abstract planning state space (Edelkamp 2001) with states $S_A \subseteq R$. Abstractions of operators $o = (P, A, D)$ are defined as $\phi(o) = (P \cap R, A \cap R, D \cap R)$. Multiple pattern databases are composed based on a partition $AP = R_1 \cup \ldots \cup R_l$ and induce abstractions $\phi_1, \ldots, \phi_l$ as well as lookup hash tables $\mathrm{PDB}_1, \ldots, \mathrm{PDB}_l$. Two pattern databases are additive, if the sum of the retrieved values is admissible. One sufficient criterion is the following. For every pair of non-trivial operators o_1 and o_2 in the abstract spaces according to ϕ_1 and ϕ_2, we have that preimage $\phi_1^{-1}(o_1)$ differs from $\phi_2^{-1}(o_2)$. For pattern database addressing we use a multivariate variable encoding, namely, SAS$^+$ (Helmert 2004).

The Algorithm

When including both incremental hashing and incremental pattern addressing into a IDA* search routine we obtain the recursive procedure IDA*-DFS as shown in Figure 1. Here, $o = (P, A, D) \in \mathit{applicable}(S)$ means $P \subseteq S$, $\mathit{doMove}(o, S)$ denotes $(S \backslash D) \cup A$ and $\mathit{undoMove}(o, S)$ equals to $(S \backslash A) \cup D$. For the ease of presentation, we assume disjoint pattern databases $\mathit{PDB}_1, \ldots, \mathit{PDB}_l$ according to the state abstractions functions $\phi_1, \ldots, \phi_l$. Disjoint pattern databases refer to different index sets. In some cases, disjoint pattern databases are additive, i.e., their lookup values can be added preserving admissibility. This depth-first traversal is invoked for each IDA* iteration with the initial state. The hash addresses for the original state space and the ones for the pattern databases are computed incrementally. The current threshold Θ, the next threshold Θ', the pattern address h_i for S in PDB_i, and the hash address h for S, are kept as global variables. To restore the state and its hash values after the subroutine call, we restore the global information in backtracking fashion. If a goal state is established, the plan is found on the recursion stack. Termination is omitted for notational convenience.

The above algorithm is optimal and complete and generalizes to many state-space search problems like the $(n^2 - 1)$-puzzle or Rubik's cube.

Consider a state space problem in vector representation, so that each state S in the space $\mathcal{S}$ can be expressed as a vector $(S_1, \ldots, S_k)$ with $S_j \in D_j$, $j \in \{1, \ldots, k\}$. Let $o = (S, S') \in \mathcal{O}$ be the applied operator, $I(o)$ the set of indices which change when applying o; $I_{\max} = \max_{o \in \mathcal{O}} |I(o)|$ and $D_{\max} = \max_{1 \leq i \leq k} |D_i|$. We obtain the following result.

Theorem 1 *Combined incremental state and pattern addressing is available in time*

```
Procedure IDA*-DFS
  for each $o \in applicable(S)$
    $h \leftarrow (h + inc(o))\ mod\ q$
    if not $(lookup(h, S))$
      $insert(h, S)$; $H \leftarrow 0$
      for each $i \in \{1, \ldots, l\}$
        $h_i \leftarrow h_i + inc(\phi_i(o))\ mod\ q$
        $H \leftarrow H + PDB_i(h_i)$
      $S \leftarrow doMove(o, S)$; $g \leftarrow g + 1$
      if $(g + H \geq \Theta)$ $\Theta' \leftarrow \min\{\Theta', g + H\}$
      else $IDA^*\text{-}DFS(S)$
      for each $i \in \{1, \ldots, l\}$
        $h_i \leftarrow h_i - inc(\phi_i(o))\ mod\ q$
      $S \leftarrow undoMove(o, S)$; $g \leftarrow g - 1$
    $h \leftarrow (h - inc(o))\ mod\ q$
```

Figure 1: IDA* with incremental state and pattern addressing.

1. *$O(|I(o)| + \sum_{i=1}^{l} |I(\phi_i(o))|)$ using an $O(kl)$-sized table. In case of* disjoint pattern databases *with different index sets, size $O(k)$ is sufficient.*
2. *$O(l)$; using an $O(l \cdot (k \cdot D_{\max})^{I_{\max}})$-sized table.*

Proof: Let $h(S) = \sum_{i=1}^{k} S_i M_i\ mod\ q$ be the hash function in $\mathcal{S}$, with $M_1 = 1$ and $M_i = |D_1| \cdot \ldots \cdot |D_{i-1}|$ for $1 < i \leq k$. Let $h_i(\phi_i(S)) = \sum_{j=1}^{k} \phi_i(S_j)\phi_i(M_j)\ mod\ q$ be the hash function for addressing pattern database *PDB*$_i$, $1 \leq i \leq l$, with ϕ_i mapping $S = (S_1, \ldots, S_k)$ to $\phi_i(S) = (\phi_i(S_1), \ldots, \phi_i(S_k)) \in \phi_i(D_1) \times \ldots \times \phi_i(D_k)$, $\phi_i(M_1) = 1$ and $\phi_i(M_j) = |\phi_i(D_1)| \cdot \ldots \cdot |\phi_i(D_{j-1})|$ for $1 < j \leq k$. In the first case we store $\phi_i(M_j)\ mod\ q$, for $1 \leq i \leq l$, $1 \leq j \leq k$. In the second case we precompute the values *inc*(o) and $inc(\phi_i(o)) = \sum_{j \in I(\phi_i(o))} -\phi_i(S_j)\phi_i(M_j) + \phi_i(S'_j)\phi_i(M_j)\ mod\ q$ for all possible $o = (S, S')$, $i \in \{1, \ldots, l\}$ with $\phi(o) = (\phi(S), \phi(S'))$. Once more, the number of possible operators is bounded by $O((k \cdot D_{\max})^{I_{\max}})$.

Partial Search

The idea of erroneous dictionaries was exploited in *Bloom filters* (Bloom 1970). It is also referred to as *bit-state hashing*. A Bloom filter is a bit vector HT of length m, together with r independent hash functions $h_1(x), \ldots, h_r(x)$. Initially, HT is set to 0. To *insert* a key x, compute $h_i(x)$, for all $i = 1 \ldots r$, and set each $HT[h_i(x)]$ to 1. To *search* a key, check the status of $HT[h_1(x)]$; if it is 0, x is not present in the dictionary, otherwise continue with $HT[h_2(x)], HT[h_3(x)]$, etc. If all these bits are set, report that x is in the filter.

Like *single* and *double bit-state hashing* (Holzmann 1998), *hash compaction* (Stern & Dill 1996) aims at reducing the memory requirements for the state table. It stores a compressed state descriptor in a conventional hash table instead of setting bits corresponding to hash values of the state descriptor in a table of bits. If the hash address and the compression are calculated independently from the state, the same compacted state can occur at different table locations.

By implementing the table of already visited nodes in IDA* with bit-state hashing or hash compaction, we establish *Partial IDA** (Hüffner *et al.* 2001). Since neither the predecessor nor the f-value are present in the state representation, in order to distinguish the current iteration from the previous ones, the bit-state table has to be re-initialized in each iteration of IDA*. Refreshing large bit-vectors is fast in practice, but for shallow searches with a small number of expanded nodes this scheme can be improved by invoking ordinary IDA* with a transposition table for smaller thresholds and by applying bit-vector exploration in large depths only. As collisions are statistically unlikely, optimality is often preserved in practice.

If the updates of the state vector in *doMove* and *undoMove* are $O(1)$, so is the time for generating one successor in the search tree of Partial IDA*. The time offset in running ordinary IDA* with transposition tables are based to the additional efforts needed for storing and retrieving states in the hash table and the conflict resolution strategy that is applied.

Experiments

For the case study, we have implemented incremental hashing for Partial IDA* in MIPS (Edelkamp 2003), since this was the only available system that supports pattern database search. A state $S \subseteq AP = \{p_1, \ldots, p_n\}$ in MIPS is represented as a bitvector $b_1, \ldots, b_n$, where $b_1 = true$ if and only if $p_i \in S$. In memory, this vector is encoded as an array of subvectors $c_1, \ldots, c_m$, where $m = \lceil n/w \rceil$ and w denotes the bit size of a memory cell. In difference to the above setting, the global hash function value is not calculated bit-wise, but word-wise. A bottleneck is the application of operator's effects. One approach is to apply a logical bit operation to the corresponding subvector for each fact. This leads to a runtime of $O(|A| + |D|)$ to apply the effects on the state.

We can improve this runtime with *combined effects*. Let set $\mathit{aff}^+ = \{\lceil i/w \rceil \mid p_i \in A\}$ and set $\mathit{aff}^- = \{\lceil i/w \rceil \mid p_i \in D\}$ denote the lists of subvector indices affected by the positive and negative effects of an operator. Based on these definitions, we pre-calculate the sets

$$m^+ = \{(i, \sum_j 2^{j\ mod\ w}) \mid i \in \mathit{aff}^+, \lceil j/w \rceil = i, p_j \in A\}$$

$$m^- = \{(i, \overline{\sum_j 2^{j\ mod\ w}}) \mid i \in \mathit{aff}^+, \lceil j/w \rceil = i, p_j \in D\}$$

Applying the operator's effects is then reduced to setting $b_i \leftarrow b_i \& m$ for each $(i, m) \in m^-$ and $b_i \leftarrow b_i | m$ for each $(i, m) \in m^+$, where $'\&'$ and $'|'$ denote bitwise AND-/OR-operators. As an example consider $w = 16$, $A = \{8, 14, 20\}$ and $D = \{18, 24, 34\}$. For this case, we have $\mathit{aff}^+ = \{1, 2\}$, $\mathit{aff}^- = \{2, 3\}$, $m^+ = \{(1, 0010000010000000), (2, 0000000000001000)\}$, and $m^- = \{(2, 1111111011111101), (3, 1111111111111101)\}$.

Using combined effects, the runtime reduces to $O(|\mathit{aff}^+| + |\mathit{aff}^-|)$, a value that is often smaller than the number of pattern databases. By precalculating m^+ and m^-, we also save the $|A| + |D|$ divisions and bit-shifts to determine the subvector and bitmask for setting or deleting a fact.

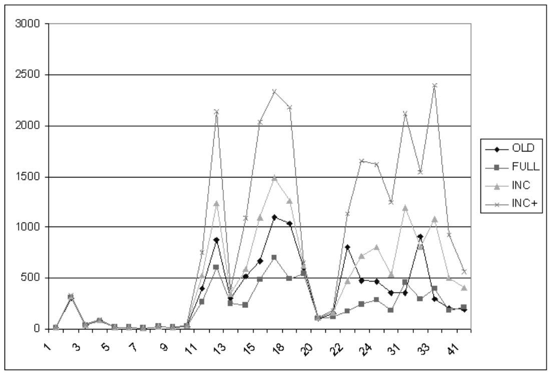


Figure 2: Results for the Pipesworld domain.

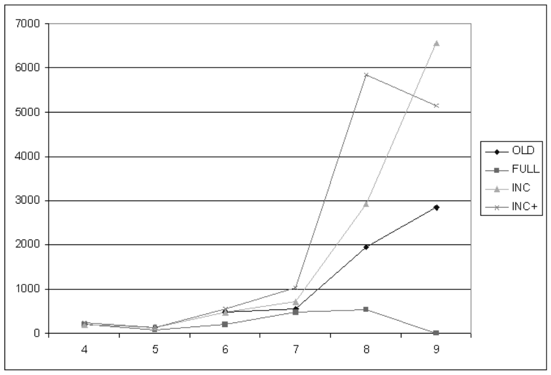


Figure 3: Results for the Blocksworld domain.

Experiments were performed on a Linux-based PC with a 1.8GHz CPU and 1 GB of main memory. The time limit was set to 120 minutes. Figures 2 and 3 show the results for all solved instances of the *Pipesworld* and *Blocksworld* domains, which were also used in the 2000 and 2004 international planning competition. The x-axes denote the problem number from the respective domain, which usually rises along with the problem size. We measure the number of expanded nodes per second, using the original hash function (OLD), the non-incremental bit-wise hashing (FULL) and the incremental hash function without (INC) and with combined effects (INC+).

As a first observation, we see that in most cases we get the best result, when incremental hashing is used. Even if no combined effects are used, there is a significant improvement over OLD and FULL. Moreover, the use of combined effects leads to even better results in most cases. In some cases, the use of incremental hashing more than doubles the the exploration speed and combined effects give further improvements. The overall poor results in column FULL emphasize the effectiveness of incremental hashing. In particular, the curve testifies that the gains in INC and INC+ can be attributed to the incremental hashing scheme rather than to the simplicity of the underlying hash function. Also, incremental hashing allows us to solve more instances within the given time limit compared to the old implementation. This is the case for the instances 16, 28 and 40 of the pipesworld domain. To summarize, incremental hashing outperforms the old hashing approach in every aspect.

Conclusion

Incremental hashing reduces the time complexity to compute the hash value of a given state from an expression linear in the state description length to a mere constant. In action planning, incremental hashing for address computations in the original and abstract state spaces significantly speeds up plan generation and can help to solve problems that cannot be solved without. Incremental hashing is available for most search problems in vector representation, including constraint-satisfaction. Incremental state hashing is particularly important for software model checking, where huge state vectors with only very little changes are to be stored. Here, patterns are devised by data or predicate abstraction.

References

Bloom, B. 1970. Space/time trade-offs in hashing coding with allowable errors. *Communication of the ACM* 13(7):422–426.

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(4):318–334.

Edelkamp, S. 2001. Planning with pattern databases. In *European Conference on Planning (ECP)*, 13–24.

Edelkamp, S. 2003. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Research (JAIR)* 20:195–238.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 161–170.

Holzmann, G. J. 1998. An analysis of bitstate hashing. *Formal Methods in System Design* 13(3):287–305.

Hüffner, F.; Edelkamp, S.; Fernau, H.; and Niedermeier, R. 2001. Finding optimal solutions to Atomix. In *German Conference on Artificial Intelligence (KI)*, 229–243.

Karp, R. M., and Rabin, M. O. 1987. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development* 31(2):249–260.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Reinefeld, A., and Marsland, T. A. 1994. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(7):701–710.

Stern, U., and Dill, D. L. 1996. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, 81–90. Shaker Verlag, Aachen.

Taylor, L. A., and Korf, R. E. 1993. Pruning duplicate nodes in depth-first search. In *National Conference on Artificial Intelligence (AAAI)*, 756–761.

A Decision-Theoretic Scheduling of Resource-Bounded Agents in Dynamic Environments

Simon Le Gloannec, Abdel-Illah Mouaddib
GREYC-CNRS
Bd Marechal Juin, Campus II, BP 5186
14032 Caen cedex, France
{**slegloan, mouaddib**}**@info.unicaen.fr**

François Charpillet
LORIA
BP 239
54506 Vandœuvre-lès-Nancy, France
charp@loria.fr

Introduction

Markov Decision Processes (MDPs) provide a good, robust formal framework for modeling a large variety of stochastic planning and decision problems, but they are unsuitable to realistic problems, in particular, to solve problems in rapidly changing environments. The existing approaches use Markov decision processes to produce a policy of execution for a static set of tasks; in a changing environment (i.e. with an evolving set of tasks), a complete computation of the optimal policy is necessary. We consider a queue of tasks that can change on-line. Potential application of this approach would be an adaptive retrieval information engine (Arnt *et al.* 2004).Our main claim is that it is possible to dynamically compute good decisions without completely calculating the optimal policy. Similar approaches have been developed to deal with MDPs with large state spaces using different decomposition techniques (Boutilier, Brafman, & Geib 1997; Parr 2000). A similar dynamic resource allocation problem has been developed in (Meuleau *et al.* 1998). A non dynamic approach has been developed for robots in (Schwarzfischer 2003).

It has been shown in (Mouaddib & Zilberstein 1998) that it is possible to find an optimal policy for this kind of problem. This optimal solution suffers from a lack of flexibilty to cope with changes in a dynamic environment. We develop an approach which provides more flexibility for MDPs to deal with dynamic environments. This approach consists of two steps. The first step consists of an off-line preprocessing of tasks and the compilation of policies for all possible available resources. The second step is a quick on-line approximation of the policy of executing the current task given the current state of the queue.

Problem Statement

Description

We consider an autonomous agent which has the capability of performing different kinds of stochastic tasks $T_1, T_2, \ldots, T_t$ progressively. A **progressive task** T is executed stage by stage, and it can be stopped after any stage. A quality is associated with each stage of task T. The duration Δr of execution of each stage is uncertain. The utility of a task is the sum of the quality of the executed stages. This formalism is described (Mouaddib & Zilberstein 1998) in details. For each type of task T_n we compute a **task performance profile** $f_n(r)$ that corresponds to its local expected value(see Figure 4). It represents what the agent should expect to gain if a certain task is being accomplished with a quantity r of resources.

Given an ordered list composed by several instances of these progressive tasks (for example $L=\{T_3, T_1, T_3, T_1, T_1, T_2, T_1, T_3, T_2, T_3, T_2\}$), the agent must perform some of them before a fixed deadline D to maximize the global utility.The queue of tasks evolves **dynamically**, during on-line phase. Tasks can appear or disappear everywhere (see Figure 1). Our goal then is to find an effective way to decide quickly if it is desirable to continue with the current task or to change to another one. This local decision depends on the prediction of the **expected value** of the remaining tasks in the list.

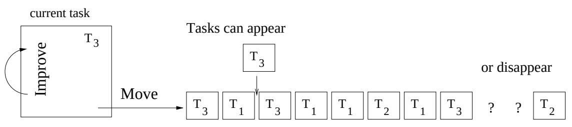


Figure 1: Dynamic changes in the task list

The decision problem We are facing to a resource allocation problem: we consider time as a **resource** r, which is **limited** by D. The duration of execution of each task is **uncertain**, resource consumption is **uncertain**. The agent has to allocate some resources for the current task, and some for the remaining tasks. Differently speaking it must **decide** if it is preferable to continue its work on the current task, or to give it up and switch to another task, by taking into account the state of current task, the list of remaining tasks, and the avalaible resources. Then, our resources allocation problem becomes a decision problem.

We present the formalism we use to solve our problem in a non-dynamic environment in the next paragraph, and in the next section we will explain how to cope with changes in the task list.

An MDP controller

At each step, the agent has to make a decision about continuing the improvement (**I**mprove) of the current task or abandoning it in favor of the next task (**M**ove). This decision depends on the current state of the decision process. In other words, it depends on the current state and the available resources. Consequently, the decision process respects the Markov property. We could deploy an MDP for the whole list of tasks, but we do not. The basic idea is to compute the policy of executing the current task. This policy $\Pi_{T,L}$ depends on L, the list of remaining tasks, and T the current task. As long as L remains unchanged, the agent follows $\Pi_{T,L}$. If L changes, it updates the current policy of the current task to $\Pi_{T,L_{new}}$. When the agent moves to a new task T_{new}, it assesses the current list of remaining tasks L_{new} and then it derives a new policy $\Pi_{T_{new},L_{new}}$.

Formal Framework An MDP is a tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{T}, Rew\}$ where $\mathcal{S}$ is a set of states representing the amount of remaining resources r, Initially, $s_0 = [D]$ where D is the deadline. Terminal states represent all the situations where r has been fully elapsed $s = [r < 0]$. $\mathcal{A}$ is a set of actions $\{\mathbf{M}ove, \mathbf{I}mprove\}$ (see Figure 3). The **M**ove action is deterministic: the agent moves from the current task and examines the next one in the dynamic list. The second action is stochastic, **I**mprove consists in spending a certain quantity of resources Δr in the next stage. $\mathcal{T}$ is a set of transitions, Rew is a reward function. In the following, we define what $\{\mathcal{S}, \mathcal{A}, \mathcal{T}, Rew\}$ means in our context. The states represent resources remaining, the actions are **I**mprove and **M**ove, and the reward is the accumulated quality for all improvements previously achieved in the task. The transitions are given in the description of each progressive task type.

$$\Pr(s' = [r - \Delta r] | s = [r], \mathbf{I}) = P_{Stage}(\Delta r) \tag{1}$$

$$\Pr(s' = [r < 0] | s = [r], \mathbf{I}) = \sum_{\Delta r > r} P_{Stage}(\Delta r) \tag{2}$$

Policy When the agent starts to execute a task T, it computes a local policy $\Pi_{T,L}$ where L represents the list of remaining tasks. Since the list L remains unchanged, it follows its policy $\Pi_{T,L}$ (see Figure 2). As soon as L changes to L_{new}, the agent has to change the local policy $\Pi_{T,L}$ to $\Pi'_{T,L_{new}}$. To do so, the computation of $\Pi_{T,L_{new}}$ is based

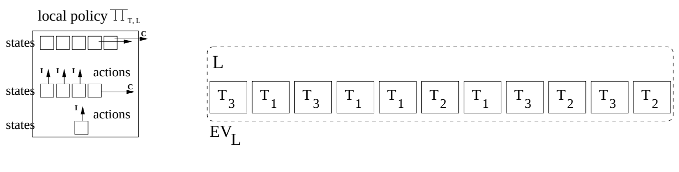


Figure 2: A local policy

on a quick computation of the new expected values of actions **M** and **I**. We describe below how those functions are computed.

Value function In order to compute the policy, we use a value function $V(s)$ based on the Bellman equation:

$$\begin{cases} V(s = [r \geq 0]) = Rew(s) + \max\limits_{\mathbf{A}} \sum\limits_{s'} \Pr(s'|s, \mathbf{A}).V(s') \\ V(s = [r < 0]) = 0. \end{cases} \tag{3}$$

Policy $\Pi_{T,L}$ is local, therefore we estimate the value of the states after a **M**ove action with a second value function that we denote as EV. EV_L is the value the agent can expect to gain if it accomplishes tasks in L with r resources. $V_{T,L}$ is the expected value of achieving task T taking into account tasks in L. In our context, the Bellman equation becomes: $V_{T,L}(s = [r]) = Rew(s) +$

$$\max_{\mathbf{A}} \begin{cases} EV_L(s = [r]) & \text{if } \mathbf{A} = \mathbf{M} \\ \sum\limits_{s' = [r \geq 0]} \Pr(s'|s, \mathbf{I}).(V_{T,L}(s')) & \text{if } \mathbf{A} = \mathbf{I} \end{cases} \tag{4}$$

where $s' = [r - \Delta r]$ represents a possible state after the improvement. Equation 4 needs to be solved as soon as the queue L changes to L_{new}. This leads to a computation of $EV_{L_{new}}$ and $V_{T,L_{new}}$. We compute first $EV_{L_{new}}$. Then, we obtain the new local policy $\Pi'_{T,L_{new}}$ by computing the $V_{T,L_{new}}$ function using a backward-chaining algorithm on all the states in T. Figure 3 represents the mechanism of action selection. In the current state the agent has 176 resources left. The dotted rectangles represent the remaining tasks in the queue L. Now, the problem is to compute the

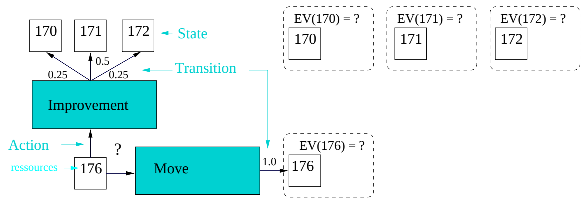


Figure 3: Action selection

Expected Value EV_L. It is possible to compute the exact EV_L of an optimal policy $\Pi^*_{T,L}$. We just have to use the Bellman equation on all the possible future states in L with a backward chaining algorithm. But with this method, the generated MDP is very large. But if we add a task in the middle of this linear graph, we must re-deploy the Bellman equation. This solution is not convenient if tasks are often inserted in the list i.e. if the environment changes. We can still keep the MDP model for the local task, but we must find another way to compute EV_L. We must evaluate the Expected Value for the rest of the plan quickly. Therefore we have to sacrifice optimality.

Rather than computing EV_L using a global MDP, we divide this process into two phases. An off-line phase, where we compile performances profiles for each type of task (see Figure 4), and an on-line phase, where we recompose dynamically the EV_L function (see Figure 5) when it is necessary.

In fact, we are faced with an MDP decomposition problem. We create a local MDP for each type of tasks, and we compute policies for local $MDPs$ $\Pi_{T,\emptyset}$ (policy of executing T assuming $L = \emptyset$). The local policies are represented by performance profiles. We recombine them on-line to obtain an approximate $EV_L \simeq recomposition_{T' \in L}(V_{\Pi_{T',L}})$.

Task performance profile construction The first phase consists of the computation and the storage of the performance profile function for each task type. The performance profile function $f_n(r)$ corresponds to the exact expected value if we have r resources to spent in the task of type

T_n.We consider that this task is independent from the others, and we compute a policy for a local MDP, without the **M**ove action. This means that we assume $L = \emptyset$. Differently speaking, $f_n(r) = V_{\Pi_{T',\emptyset}}([r])$. This computation is quick. The local MDP has few states, and each state is evaluated once. It only depends on the number of stages in the task, and also on the maximum amount of resources that the agent can spent in the tasks. Now, we have to combine all these functions f_n to find an EV_L function for a list composed of tasks of type $\{T_1, T_2, \ldots, T_n\}$. This function must be a good approximation of the exact EV_L.

Dynamic operation

Principle

This section is divided into two phases: the off-line phase, where we compute the performance profile $f_n(r) = V_{\Pi_{T',L}}([r])$ for each task, and the on-line phase, where the agent recomposes the expected value function EV_L.

Off-line: The task pre-processing The performance profile functions $f_n(r)$ of each task are increasing functions in r. Note that the more time the agent spends on a task, the higher the expected reward will be. These curves are also bounded by a maximum, which corresponds to the reward if the task is completely achieved. The curves give us several information. On the one hand we know the expected value for this task, while on the other, we know the quality cost ratio, i.e. $f_n(r)/r$. The first phase of the pre-processing consists of finding the best quality cost ratio among all tasks.

$$max_{n,r} f_n(r)/r \tag{5}$$

Then we isolate the part of the curve which precedes this point, as figure 4 shows. We start again to seek the second best quality cost ratio, until all the curves are processed. We finally store the points and slices of curves $c_{i,j}$ in a table R_T (cf algorithm 1). $c_{i,j}$ is the j^{th} slice of curve f_i.

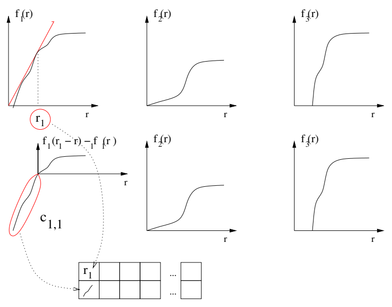


Figure 4: Finding the best ratio quality/cost

On-line: Expected Value function reconstitution It is the second phase. All the next computations are done at run time. We recompose the EV_L function with all the information that we stored during the off-line phase $\{f_1, \ldots, f_t\}$, R_T, and with the list. The method is simple (see Algorithm 2). We have a list of tasks = $\{T_3, T_1, T_3, T_1, T_1, T_2, T_1, T_3, T_2, T_3, T_2\}$, and we recompute the expected value for all possible values of r, which represents the remaining time at a given moment. Algorithm 2 is illustrated in figure 5. Basically, we want that

Algorithm 1 Task pre-processing

```
Require: ${f_1; \ldots; f_t}, R_T = \emptyset$
 1: while $\forall i \le t, \exists r, f_{T_i}(r) > 0$ do
 2:     $r'_{t'}, f_{t'} = max_{i,r} f_i(r)/r$
 3:     $R_T \leftarrow R_T \cup \{[r'_{t'}, f_{t'}(r \le r'_{t'}]\}$
 4:     $f_{t'}(r) = f_{t'}(r - r'_{t'}) - f_{t'}(r'_{t'})$
 5: end while
Ensure: $R_T$
```

the agent maximizes its future rewards, i.e. its EV_L. Thus, the EV_L curve is recomposed incrementally by adding each slice of curve in R_T until all available resources has been fully elapsed. This problem is similar to the knapsack problem. We start to add the slice of curve that maximize the ratio quality/cost. We add as many slices as the number of tasks of that type in L allowed by the remaining resources. Then, we add the slice of curve that corresponds to the second ratio quality/cost in R_T. This processing continues until all slices of curve $c_{i,j}$ in R_T have been added or the resources have been fully elapsed. In the example L contains four instances of the task T_1. In the previous section, i.e during the off-line phase, we found that the first part of the task of type T_1 gives a maximum quality cost ratio for r_1 resource. Consequently, if we have only r_1 resources left, it will be better to spent them on a task T_1. Thus, we start by adding one time the corresponding slice of the curve $f_1(0 \le r \le r_1)$. We are not sure that the exact EV_L corresponds to the slice of curve we add, it is just an approximation. But we are sure that we can not expect[1] less than this slice of curve, i.e. the approximation we make is a lower bound for the exact expected value function. If it has r resources $r_1 \le r \le 2 \times r_1$, it can do the first part of a task T_1, and starts the second task T_1. Therefore, we add the same slice of curve on the top of the first one (see Figure 5). In our example, we have four tasks of type T_1 in the list, so if we

Algorithm 2 Reconstitution of the Expected Value function (approximation) EV_L

```
Require: $R_T, L$
 1: $threshhold = 0$
 2: $r = 0$
 3: while $R_T \neq \emptyset$ do
 4:     $(r', f_{t'}) \leftarrow first(R_T)$
 5:     remove $first(R_T)$ from $R_T$
 6:     $n$ = number of task $T_t \in L$
 7:     for $i = r,\ i \le n \times r', i^{++}$ do
 8:         $m = i \ mod \ r'$
 9:         $q = i \ div \ r'$
10:         $EV_L(i) = m \times f_{t'}(r') + f_{t'}(q) + threshhold$
11:     end for
12:     $r = r + n \times r'$
13:     $threshhold = EV_L(r)$
14: end while
Ensure: $EV_L$
```

[1]Note that it is an expectation. During the execution of this task we can have less reward than what we expected.

have more than $r > 4 \times r_1$ resources, the agent can expect to start four times T_1. With the rest ($r - 4 * r_1$) it can expect to start another task. In R_T, T_2 is the best task to start after T_1. We add the corresponding slice of curve to EV_L between $4 \times r_1$ and $4 \times r_1 + r_2$. We continue adding slices of curve while R_T is not empty. If the agent has more than R_{max} resources then it is certain to achieve all the tasks, with all their improvements. The approximation of EV_L is finished.

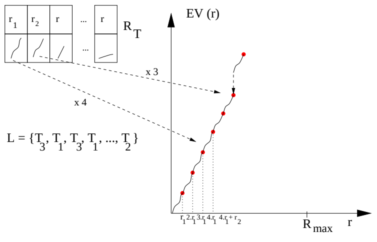


Figure 5: Reconstition of $EV_L(r)$

How do we use this function in practice

The dynamic on-line approximation In the preceding section, we explained how to reconstitute the expected value. However in practice we do not need to know all the values of $EV_L(r)$. Let D -r_{now} be the resources remaining to complete the mission. Moreover, we can use only a maximum amount of resources r_{max} for the current task. In fact, we just compute EV_L for all r in $[D$ -r_{now} -r_{max}, D -$r_{now}]$. We made this reconstitution in order to compare $EV_L(r)$ obtained by our approach with that obtained while solving the whole MDP for all the remaining tasks in the list. The complexity of computation is proportional to the number of elements in R_T. Thus, the computation of $EV_L(r)$ is quick and is more suitable to the dynamicity.

Analysis

In this section, we make a comparison between EV_{exact}, EV_{dyna}, and $EV_{pw-linear}$. This comparison concerns the time needed to compute each of them and the error made by our approach.

Complexity Comparison To compute EV_{exact} we develop a set of states $\mathcal{S} = \{s = [r, imp, T]\}$, r are the remaining resources, imp the quality of the last improvement made, and T is the task in the queue. r is considered discreet. We use a backward chaining algorithm using the Bellman equation, therefore each state is evaluated only once. The complexity of EV_{exact} is linear in the number of states $\#\mathcal{S}$. But the state space is huge. $m(EV_{exact}) = \#\mathcal{S} = R_{max} \times \prod_{T_i \in L} \#imp_{T_i}$. m is a mesure of complexity. $m(EV_{dyna}) = \sum_{R_T} \#c_{i,j} \times \#T_i$ where $c_{i,j}$ is a slice of the T_i performance profile curve, and $\#T_i$ the number of times where T_i appears in L. $m(EV_{dyna})$ is linear in the size of R_T and L. There are much fewer slices in R_T than there are states in $\mathcal{S}$. Thus, $m(EV_{dyna}) \leq m(EV_{exact})$. For a queue of 100 tasks, EV_{exact} takes several minutes, and EV_{dyna} takes less than one second. More results can be found on *http://users.info.unicaen.fr/˜slegloan/thesis/.*

Value Comparison Our approximation EV_{dyna} is just a lower bound of the EV_{exact} function. If all the tasks executions where deterministic, EV_{dyna} and EV_{exact} would be equal on some specific points, at the ”end” of each slice (fig 6).

$$\forall r, EV_{dyna}(r) \leq EV_{exact}(r). \qquad (6)$$

Unfortunatelly, our approximation is just a lower bound

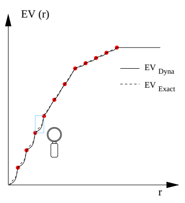


Figure 6: Deterministic Tasks

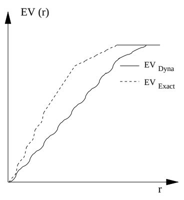


Figure 7: With Uncertainty

of what the agent should expect to gain in the worst case. We intend to mesure the difference between EV_{dyna} and EV_{exact} in the future, in order to fill the gap betwenn the two curves.

Conclusion and future works

We have presented a robust solution that copes with uncertainty due to dynamicity of environment in stochastic planning problems. The Markovian approach allows us to cope with uncertainty. The progressive approach allows us to adapt the decision in a dynamic environment. Our work combines two approaches: Progressive tasks and decomposition of large MDPs. For the future, we intend to add multiple limited resources, like energy. We would also like to extend this approach to include more specific spatial and temporal constraints. This model can be adapted to a large set of problems in a dynamic and uncertain environment like robotic applications, and information retrieval agent.

References

Arnt, A.; Zilberstein, S.; Allan, J.; and Mouaddib, A. 2004. Dynamic composition of information retrieval techniques. In *Journal of Intelligent Information Systems*, 23(1):67–97.

Boutilier, C.; Brafman, R.; and Geib, C. 1997. Prioritized goal decomposition of markov decision processes: Toward a synthesis of classical and decision theoretic planning. In *IJCAI 97*, 1156–1163.

Meuleau, N.; Hauskrecht, M.; Kim, K.; Peshkin, L.; Kealbling, L.; Dean, T.; and Boutilier, C. 1998. Solving very large weakly coupled markov decision processes. In *UAI-98*.

Mouaddib, A., and Zilberstein, S. 1998. Optimal scheduling for dynamic progressive processing. In *ECAI-98*.

Parr, R. 2000. Flexible decomposition algorithms for weakly coupled markov decision process. In *UAI-00*.

Schwarzfischer, T. 2003. Quality and utility - towards a generalization of deadline and anytime scheduling. In *ICAPS 2003*, 277–286.

Stream Processing Planning

G. Ya. Grabarnik, Z. Liu, A. Riabov
IBM T.J. Watson Research Center,
19 Skyline Dr., Hawthorne, NY 10532, USA
{genady,zhenl,riabov}@us.ibm.com

Abstract

We investigate planning approaches to the problem of composing distributed data stream processing applications. Workflow composition problems have drawn attention of researchers in several areas of computer science. AI planning approaches have been recently applied for solving workflow composition problems in web services, semantic grid, business process modeling and automated installation. The requirements of stream processing planning applications include the ability to express the functional dependence between attributes of incoming and outgoing streams and the ability to share a data stream between multiple subscribers. We extend the well-known PDDL formalism and propose a new planning model that is specifically targeted for describing stream processing planning requirements. We present our initial analysis of decidability and complexity of the problem, and discuss planning methods.

Overview

Today powerful computers and high-bandwidth communication are becoming increasingly available, stimulating the growth in the use of high-performance distributed computing. At the same time new software development tools enable large- scale distributed component-based software architectures. In this environment making the choice between the available components and services and establishing the interconnections between the components is a complex and tedious task. The difficulty of this task is preventing many categories of users from taking advantage of otherwise easily available computational power. The end users would prefer to specify the desired outcome, and let the system make the choices and establish necessary connections automatically. Automatic composition problems arising in this context are naturally related to planning, since a sequence of decisions must be made in order to choose and interconnect components that form the processing system.

In recent work planning methods originating in AI literature have been successfully applied to web service composition, automated software installation, and deployment of component-based software (for example see (Doshi *et al.* 2004), (Kichkaylo, Ivan, & Karamcheti 2003)). However, these models use very simple rules for connecting the components. Typically in these models the exact matching between the symbolic descriptions of input and output data types is the sufficient condition for linking the output of one component to the input of another.

The problem we describe in this paper arises in stream processing applications, where the volume of the data being processed is too large to be stored, and therefore the information must be processed on the fly. Examples of such applications include video processing, streaming databases and sensor networks. We consider a compositional programming model that allows building stream processing applications from a set of reusable components. Each component exposes one or more input port, and one or more output port. Once each input port of a component is connected to a stream, the component will produce one output stream for each of the output ports, by filtering, annotating, or otherwise processing and transforming the information it receives. Once the output stream is created any number of components can read from it, provided that the input requirements are compatible with the stream characteristics. In this model at most one stream can be connected to one input port and therefore each stream can have only one writer and any number readers. A formal definition of stream processing semantics can be found in (Klein, Rumpe, & Broy 1996).

We will say that the streams that arrive from the outside are the *primal streams*. The set of available primal streams defines the initial state for planning. The components can transform both primal and derived streams to produce new *derived streams*. Each primal or derived stream can be described by a set of properties – stream *tags*. We assume that the tags corresponding to the primal streams are fixed and given. Each component is associated with a formula for computing output stream tags based on input stream tags. The goal of stream processing can be formulated as a logical expression that specifies the desired characteristics of the final output stream. The solution of the planning problem is a DAG of components linked by streams (see Figure 1).

Although stream planning problem is very similar to the traditional planning problems solved by domain-independent solvers, we encountered the following difficulties during during our attempts to formulate this problem in PDDL:

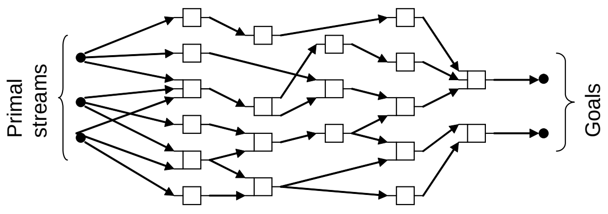


Figure 1: Example of a stream processing application graph.

- Stream planning problems are inherently symmetric, which is not easily detectable if only the PDDL formulation is available. As the result, many currently available planners waste time evaluating different equivalent assignments of stream objects to stream variables.
- In stream processing planning each action corresponds to placing a new component in the workflow. The action creates new streams, but does not modify the existing streams. Since multiple instances of the same component may be used in the plan, the action should be written in a way that always assigns outputs to new streams and avoids rewriting of the already initialized ones. While with some effort this can be done in PDDL, the planners lose performance because of the inefficient encoding.

In this paper we introduce a general formalism for describing the stream processing planning problem. This formalism can also be used to model the composition of web services or other software components that adhere to the producer-consumer paradigm. Finally, we present preliminary performance comparisons and discuss future research directions.

Stream Planning Domain

We start with a strict description of generic Stream Processing Planning. We show that it is more expressive than the STRIPS model (see (Nebel 2000)). Then we show that complexity of the planning for our model changes between PSPACE and EXPSPACE. For the state variable formalism see (Jonsson & Bäckström 1998).

Types and Constants Types represent a finite tree based on the inheritance relation. Only single (not multiple) inheritance is allowed. Type $object$ is a root type for all types.

```
(:types person city address - object,
                          s - stream)
```

Constants may be of certain types. There are only finitely many constants of a specific type.

```
(:constants ernie scott dan - person)
```

Streams, Variables and Functions A Stream is special type of object. It corresponds to a set of typed constants. To reflect that stream s contains constant o we use predicate

```
has_object ( s, o) or (s and (o))
```

In the state variables representation of the world, streams correspond to an additional grouping of the state variables.

```
(:action AA
  :parameters (?in1 ?in2 ?in3 ?out1 ?out2)
  :precondition (in1 and (P1)(P2))
  :precondition (in2 and (P3)(P2))
  :precondition (in3 and (P5)(P6)(P7))
  :effect (out1 and (P4)(P6))
  :effect (out2 and (P3)(not (P2)))
  (increase (cost) 1.0)
  (increase (obj) 24.0) )
(:action CA
  :parameters (?in1 ?in2 ?in3 ?out1 ?out2)
  :precondition (in1 and (P1)(P2))
  :precondition (in2 and (P3)(P2))
  :precondition (in3 and (P5)(P6)(P7))
  :effect (out1 when in1.P4 then (P6))
  :effect (out2 when in3.P2
               then ( and (P3) (P5) )
  (increase (cost) 1.0)
  (increase (obj) 24.0) )
```

Figure 2: Sample assignment action (AA); sample action with conditional effects (CA).

We assume that all the variables are streams. For convenience we think of streams as partially grounded complete lists with enumeration corresponding to types (or we can add a new constant ϵ that would mean that a value is not specified or not important, see for example (Jonsson & Bäckström 1998). Let s_1 be a stream consisting of constants $o_1, o_2, ..., o_l$. Stream s_2 extends stream s_1 if the stream s_2 contains at least l constants $o_1, o_2, ..., o_l$ on proper type positions. Stream s_g is a grounded extension of s_1 if all possible types-placeholders in s_g are set to certain values and s_g extends s_1. We use the notion of functions and elemental arithmetic operations in the same way it is used in PDDL2.2.

Relations and Actions Relations are the fixed relations on the streams that are independent of changes in the streams. Actions have preconditions and effects (for action A we denote them by precond(A) and effects(A)). Both the preconditions and the effects are expressed in terms of streams. Each precondition is a set of expressions on the corresponding stream and relations. All preconditions are independent, and correspond to different streams. Each effect is a set of assignments of values to the stream depending on preconditions. Assignments may depend in principal on certain boolean expression. We separate the following cases: simple assignment, conditional assignment ($when\ E\ then\ S$), and boolean expression assignment containing more complex boolean expressions. All effects are independent from other effects and act on different streams.

Note that the difference with state-variable presentation is that streams contain a complete set of types, and that actions have multiple independent (different streams) preconditions and multiple independent effects (see (Jonsson & Bäckström 1998)). Since effects are independent from each other, actions may be decomposed into a set of actions with one stream output. However, since effect depend on preconditions, action could not be decomposed further.

Planning language $\mathcal{L}$ in the stream representation is a set of the stream variables $\mathcal{S}$, a finite set of fixed relations $\mathcal{R}$ and sets of constants that define their streams. Define set X of all partially grounded streams such that there exists a grounded extension x_g of $x \in X$ satisfying fixed relations R.

Planning domain in language $\mathcal{L}$ is a state transition system $\Sigma(\mathcal{S}, \mathcal{A}, \mathcal{F}, \gamma)$ satisfying

- $\mathcal{S} \subset \Pi_{x \in X} D_x$, where D_x is a range of the partially grounded stream variable x. In this case state s defined as $s = \{(x = c) \mid x \in X\}$ for $c \in D_x$.
- $\mathcal{A} = \{$ all partially grounded instances of actions that meet relations in $\mathcal{R}\}$. We say that action A is applicable to the stream set $\{s_1, ..., s_k\}$ if action A has k preconditions $\{in_1, ..., in_k\}$ and stream s_i extends precondition in_i for $i = 1, ..., k$.
- $\gamma(s, A) = \{(x = c) \mid x \in X\}$ where c is specified by assignment $c \mapsto x$ in effects (A).
- $\mathcal{S}$ is closed under γ, meaning for $s \in \mathcal{S}$ and every action A applicable to s one has $\gamma(s, A) \in \mathcal{S}$.
- $\mathcal{F}$ is a set of all functions defined by finite compositions of elemental operations on $\mathcal{S}$.

Planning goal is a triple $\mathcal{P} = (s_0, \mathcal{F}_\prime g)$ where s_0 is a set of grounded streams representing initial state in $\mathcal{S}$, $\mathcal{F}_\prime$ is a set of initial values of the functions, and the goal g is a set of expressions on the stream variables.

Planning problem is a 6-tuple $P = (\mathcal{A}, \mathcal{R}, \mathcal{F}, s_0, \mathcal{F}_0, g)$ where $\mathcal{A}$ is a set of actions, $\mathcal{R}$ is a set of fixed relations, $\mathcal{F}$ is a set of all functions defined by finite compositions of elemental operations on $\mathcal{S}$, s_0 is the initial state, $\mathcal{F}_0$ is a set of initial values of the functions, and g is a goal.

Decidability of the planning problem (see (Erol, Nau, & Subramanian 1995)). It is easy to show that planning language that contains notion of function is undecidable. Let P be a stream processing planning problem. Suppose that solution for the planning problem exists and has length l. First, we ground all possible actions in such a way that they still satisfy relations $\mathcal{R}$. Next, we can enumerate all admissible paths of length l: first enumerate all admissible paths of length 1, then, given enumeration of length $m-1$, we can enumerate all admissible paths of length m, by adding to each path every admissible grounded action. Suggestion of existing of the planning solution implies that solution maybe found. This means that stream planning problem is at most semidecidable. It also implies that finding length of the plan is decidable problem. Semidecidability follows from the fact that this model covers classical planning, and the fact that classical planing is semidecidable (see Corollary 3.1 and Theorem 3.3 in the reference above).

Complexity of the planning problem. Since our model covers classical planning, this fact and Theorem 5.7 of (Erol, Nau, & Subramanian 1995) implies that stream processing planning problem is EXPSPACE hard.

The advantage of using stream planning formulation directly, instead of constructing a PDDL formulation first, solving it, and later converting the solution back to streams and stream processing components, lies in the added efficiency that the search algorithm can gain from the additional structure present and explicitly specified in the stream planning formulation. The approach of using variables to denote input and output streams in PDDL actions may cause optimizing domain-independent planners to enumerate $O(N!)$ solutions for each unique solution of length N, due to the symmetry in assigning stream objects to variables. Detecting these symmetries in metric problems can be difficult.

Preliminary Experiment Results

We have implemented a general branch-and-bound framework for solving the problem, that allows us to experiment with different search algorithms, as well as bounds and heuristics. Branch and bound is a standard approach to solving combinatorial problems, and it have been shown to be a successful solution method for planning problems.

Planning Algorithm

Currently, the backward search (from the goal) is implemented: at each branching node a goal is chosen from the set of available nodes, and is connected to an existing primal or derived stream or to a newly placed action. If an action is placed, the input ports of the action are registered as new goals to be satisfied at the next step. The preconditions of the action in combination with the constraints on the output of the action are used to specify the new goal constraint for each of the inputs. The search tree is pruned if the best achieved total cost is exceeded. Similarly to PDDL, our implementation allows predicates and actions to have parameters. The parameters are substituted before the search.

The algorithm gains additional efficiency from precomputing pairs of commuting actions and considering only one of the two possible orderings in the pair, therefore achieving the same effect as GraphPlan (Blum & Furst 1995) does in allowing the commuting actions to be executing in parallel, extending this approach to the more general stream planning scenario. Potential conflicts (mutexes) are precomputed at the same time, and for each input port of each action a list of output ports that can be connected to it are constructed. These lists are reduced during branching according to the revised goals.

Experiment

We have conducted a number of experiments to study the performance gain due to incorporating additional knowledge in planning language. In these experiments we compare the performance of our planner to the performance of Metric-FF (Hoffmann 2003) and LPG-*td* (Gerevini, Saetti, & Serina 2004). We have chosen these planners because they have demonstrated top performance among metric solvers in the International Planning Competition (in 2002 and 2004 resp.), and were available for evaluation.

In our experiments we generated a simple instance of the stream processing planning problem, and formulated it in both PDDL and the extended planning language. We then varied the size of the example and measured the time it takes each of the planners to find the solution. We have constructed the examples that represent elementary planning problems that are likely to occur (in combinations) in practical stream processing planning scenarios. Therefore, the performance demonstrated by Metric-FF and LPG-*td* on these examples is indicative of the performance on significantly more complex problems that arise in practice.

All planners were run in sequence on the same 3.0 Ghz Pentium 4 computer with 500 megabytes of memory. For the same problem size, performance of the same planner can vary due to the randomness in problem instance generation and the random decisions taken by the planner (for example, LPG-*td* employs random restarts). Therefore we measured the average planning time on 15 randomly generated instances for each problem size. The experiments were terminated if the running time exceeded 10 minutes.

The problem instances are constructed by first generating a flow graph: a random binary in-tree rooted at the goal stream, in which nodes have 2 incoming links with probability 0.3. For each arc in this tree we assign a unique predicate. This predicate is then created as an effect on the corresponding output port of the action at the tail of the arc. The same predicate is then required at the input port of the action at the head of the arc. The predicate produced by the root is listed in the goal requirements. This assignment of predicates mimics the stream data type compatibility constraints. It ensures that the generated flow graph can be reconstructed during planning.

The leaf nodes in this tree are actions corresponding to the data sources. The data sources are the generators of the primal streams. They do not require any specific inputs and can be inserted in the plan at any time. However, while regular actions can appear in the plan multiple times, at most one instance of each data source can be included in the plan. This condition may be enforced directly by the planner, as in our implementation, or via a global predicate.

Plan	*Stream*			*Metric-FF*			*LPG-td*		
size	min	avg	max	min	avg	max	min	avg	max
5	0.07	0.08	0.11	0.03	0.08	0.26	0.33	0.42	0.66
11	0.08	0.09	0.10	0.16	10.67	56.63	0.45	1.01	2.28
13	0.08	0.09	0.12	37.95	*	*	0.78	#	#
15	0.08	0.09	0.10	46.03	*	*	0.93	12.70	19.77
19	0.08	0.09	0.12	*	*	*	1.49	19.76	25.70
25	0.09	0.11	0.30	*	#	#	4.12	12.08	23.94
30	0.10	0.12	0.21	*	*	*	12.8	29.8	57.5
35	0.10	0.11	0.14	*	*	*	329.0	*	*
50	0.12	0.13	0.14	*	#	#	#	#	#
100	0.16	0.17	0.21	#	#	#	#	#	#
500	0.52	0.55	0.63	#	#	#	#	#	#

The table above the results of our experiments. The first column contains the number of actions in the plan and the other columns contain the minimum, maximum and average running time in seconds for all planners that we tested. Column titled *Stream* corresponds to our stream processing planner implementation. In the table, "*" indicates that the solution was not found after 10 minutes, and "#" indicates that the solver terminated abnormally due to insufficient memory or other reasons. In this experiment the timeouts were often caused by thrashing that occurs when planners allocate and use more memory than is physically available.

The problems constructed for this experiment are very simple, and require no search when formulated in the extended planning language. The experiment shows that general-purpose planners cannot find the solution when problem size reaches 50. We cannot expect better performance of the planners on more realistic stream planning problems to be better, since the simple tree structures will likely appear as subgraphs in complex workflows. Therefore, the stream planning formalism can significantly improve performance on these problems.

Conclusion

In this paper we presented a new formalism for stream processing planning problems. We defined the model for describing the problem and provided initial analysis of the decidability and complexity of the model. Finally, we have performed experimental analysis and showed that significant performance gains can be obtained due to improved representation of the problem.

While the knowledge of problem structure provides performance improvements, there is also a need for more detailed investigation of planning algorithms. Stream planning problems are very similar to classical planning problems, and we think that many of the search techniques developed for traditional planning can be applied in this context.

References

Blum, A. L., and Furst, M. 1995. Fast planning through planning graph analysis. In *IJCAI-1995*, 1636–1642. Montreal.: Morgan Kaufmann.

Doshi, P.; Goodwin, R.; Akkiraju, R.; and Verma, K. 2004. Dynamic workflow composition using Markov decision processes. In *ICWS-2004*.

Erol, K.; Nau, D.; and Subramanian, V. S. 1995. Complexity, decidability and undecidability results for domain independent planning. *Artificial Intelligence* 76(1-2):75–88.

Gerevini, A.; Saetti, A.; and Serina, I. 2004. Planning in PDDL2.2 domains with LPG-TD. In *ICAPS-2004*.

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

Jonsson, P., and Bäckström, C. 1998. State variable planning under structured restrictions: Algorithm and complexity. *Artificial Intelligence* 100(1-2):125–176.

Kichkaylo, T.; Ivan, A.; and Karamcheti, V. 2003. Constrained component deployment in wide-area networks using ai planning techniques. In *IPDPS-2003*.

Klein, C.; Rumpe, B.; and Broy, M. 1996. A stream-based mathematical model for distributed information processing systems. In *FMOODS-1996*.

Nebel, B. 2000. On the compilability and expressive power of propositional planning formalisms. *Journal of AI Research* 12:271–315.

Probabilistic Temporal Planning in Prottle — Extended Abstract

Iain Little, Douglas Aberdeen, and **Sylvie Thiébaux**
National ICT Australia & Computer Sciences Laboratory
The Australian National University
Canberra, ACT 0200, Australia
name.surname@anu.edu.au

Abstract

We present a general framework for probabilistic temporal planning in which effects, the time at which they occur, and action durations are all probabilistic. This framework includes a search space that is designed for solving probabilistic temporal planning problems via heuristic search, an algorithm that has been tailored to work with it, and an effective heuristic that is based on an extension of the planning graph data structure. `Prottle` is a planner that implements this framework, and can solve problems expressed in an extension of PDDL.

Introduction

Probabilistic temporal planning is the combination of concurrent durative actions and probabilistic effects. This unification of the disparate fields of probabilistic and temporal planning is relatively immature, and presents new challenges in efficiently managing an increased level of expressiveness.

The most general probabilistic temporal planning framework considered in the literature is that of Younes and Simmons (2004). It is expressive enough to model generalised semi-Markov decision processes (GSMDPs), which allow for exogenous events, concurrency, continuous-time, and general delay distributions. This expressiveness comes at a cost: the solution methods proposed in (Younes & Simmons 2004) lack convergence guarantees and significantly depart from the traditional algorithms for both probabilistic and temporal planning. Concurrent Markov decision processes (CoMDPs) are a much less general model that simply allows instantaneous probabilistic actions to execute concurrently (Mausam & Weld 2004). Aberdeen *et al.* (2004) and Mausam and Weld (2005) have extended this model by assigning actions a fixed numeric duration. They solved the resulting probabilistic temporal planning problem by adapting existing MDP algorithms, and have devised heuristics to help manage the exponential blowup of the search space.

We present a general framework for probabilistic temporal planning, in which not only do the (concurrent) durative actions have probabilistic effects, but the action durations and discrete effect times can vary probabilistically as well. According to Mausam and Weld (2005), probabilistic planning under these relaxed assumptions goes significantly beyond their own work. We demonstrate that it is possible to achieve this level of expressiveness and yet still maintain a close alignment with existing work in probabilistic and temporal planning (Smith & Weld 1999; Blum & Langford 1999; Bacchus & Ady 2001; Bonet & Geffner 2003). The framework is implemented in a planner called `Prottle`. We demonstrate `Prottle`'s performance on a customised benchmark. This paper is based on the thesis (Little 2004), which we refer to for further details.

Probabilistic Durative Actions

```
(:durative-action jump
    :parameters (?p - person ?c - parachute)
    :condition (and (at start (and (alive ?p)
                                   (on ?p plane)
                                   (flying plane)
                                   (wearing ?p ?c)))
                    (over all (wearing ?p ?c)))
    :effect (and (at start (not (on ?p plane)))
                 (at end (on ?p ground))
                 (at 5
                   (probabilistic
                     (parachute-opened 0.9 (at 42 (standing ?p)))
                     (parachute-failed 0.1
                       (at 13 (probabilistic
                                (soft-landing 0.1
                                  (at 14 (bruised ?p)))
                                (hard-landing 0.9
                                  (at 14 (not (alive ?p))))))))))
```

Figure 1: An example of an action to jump out of a plane.

`Prottle`'s input language is the temporal STRIPS fragment of PDDL2.1 (Fox & Long 2003), but extended so that effects can be probabilistic, as in PPDDL (Younes & Littman 2004). We also allow effects to occur at any time within an action's duration. The probabilistic and temporal language constructs interact to allow effect times and action durations to vary probabilistically. For clarity, each probabilistic alternative is given a descriptive label.

Figure 1 shows an example action representing a person jumping out of a plane with a parachute. After 5 units of time, the person attempts to open the parachute. The case where this is successful has the label `parachute-opened`, and will occur 90% of the time; the person will gently glide to safety, eventually landing at time 42. However, if the parachute fails to open, then the person's survival becomes dependent on where they land. The landing site is apparent at time 13, with a 10% chance of it being soft enough for the person to survive. Alive or dead, the person then lands at time 14, 28 units of time sooner than if the parachute had opened. But regardless of the outcome, or how long it took to achieve, the action ends with the person's body on the ground.

We treat the structure of an action's possible outcomes as a decision tree, where each non-leaf node corresponds to a probabilistic event, and each leaf node to a possible outcome. As each event is associated with a delay, this structure allows for partial knowledge of an action instance's actual outcome by gradually traversing the decision tree as time progresses.

The duration of an action is normally inferred from the effects that have a numeric time, and depends on the path taken through the decision tree. The decision tree representation assumes that probabilistic alternatives occur to the exclusion of the others. Nevertheless, independent `probabilistic` events are allowed by the input language; any independence is compiled away by enumerating the possibilities.

Search Space and Algorithm

There is a well-established tradition of using the Markov decision process framework to formalise the search space for probabilistic planning algorithms. We take a slightly different approach, by formalising the search space in terms of an AND/OR graph that more closely aligns with the structure of the problem.

In the interpretation that we use, an AND node represents a *chance*, and an OR node a *choice*. We associate choice nodes with the selection of actions, and chance nodes with the probabilistic event alternatives. Each node is used in one of two different ways: for *selection* or *advancement*, in a manner similar to some temporal planners (Bacchus & Ady 2001). The rules for node succession can be summarised as: every successor of a node must either be a selection node of the same type, or an advancement node of the opposite type.

Rather than deal with an action's probabilistic branching immediately after it is selected, this search space structure is intended to be used with a 'phased' search where action selection and outcome determination are kept separate. This allows the time at which the action outcomes are known to be accurately represented, by deferring the branching until the appropriate time.

As an example, we now describe a path through such an AND/OR graph, starting from an advancement choice node. First, we choose to start an instance of the `jump` action from Figure 1, which progresses us to a selection choice node. We can now choose either to start another action, or to 'advance' to the next phase; we choose to advance, and progress to an advancement chance node. There is a current probabilistic event with alternatives `parachute-opened` and `parachute-failed`. Let us say that the parachute fails to open for our chosen path, which leaves us at a selection chance node. There are no more events for the current time, so we progress to another advancement choice node. Rather than start another action, we then choose to advance again. The next probabilistic event has alternatives `soft-landing` and `hard-landing`. Let us be nice and say that the person lands on something soft.

Using the graph structure that we have established, we define a state of the search space as a node in an AND/OR graph that is identified by a *time*, *model* and *event queue*. The time of a state is generally the same as its predecessors, but may increase when advancing from choice to chance. The model is the set of truth values for each of the propositions, and the event queue is a time-ordered list of pending events. An event can be an effect e.g. `(on ?p ground)`, a probabilistic event, or an action execution condition that needs to be checked. When the time is increased, it is to the next time for which an event has been queued. We define the *initial state* as an advancement choice state with time 0, the initial model, and an empty event queue. We define a *goal state* as any state in which the model satisfies the problem's goal.

We associate states with both lower and upper cost bounds. As the search space is explored, the lower bounds will monotonically increase, the upper bounds monotonically decrease, and the actual cost is sandwiched within an ever-decreasing interval. We say that a state's cost has *converged* when, for a given $\epsilon \geq 0$: $U(s) - L(s) \leq \epsilon$ where U is the upper bound and L the lower bound of state s. For convenience, we restrict costs to the interval $[0, 1]$. The cost of a state is just the probability of the goal being **unreachable** from it if only optimal choices are made. New states are either given a lower bound of 0 and an upper bound of 1, or values that are computed using appropriate heuristic functions. Although we only consider probability costs in this paper, the cost scheme that we describe can easily be generalised to include other metrics, such as makespan. The main restriction is that the cost function needs to be bounded.

A state's cost bounds are updated by comparing its current values with those of its successors. We use the following formulae for updating probability costs, where (1)–(2) are for choice states, and (3)–(4) are for chance states:

$$L_{\textbf{choice}}(s) \;:=\; \max(L(s), \min_{s' \in S(s)} L(s')), \quad (1)$$

$$U_{\textbf{choice}}(s) \;:=\; \min(U(s), \min_{s' \in S(s)} U(s')), \quad (2)$$

$$L_{\textbf{chance}}(s) \;:=\; \max(L(s), \sum_{s' \in S(s)} P(s')\, L(s')), \quad (3)$$

$$U_{\textbf{chance}}(s) \;:=\; \min(U(s), \sum_{s' \in S(s)} P(s')\, U(s')), \quad (4)$$

where S is the set of successors of state s, and P is the probability of s. We define the probability of a selection chance state as the probability of its probabilistic event alternative. The probability of all other states is 1.

In addition to a cost, we also associate each state with a label of either *solved* or *unsolved*. A state is labelled as solved once the benefit of further exploration is considered negligible; for instance, once its cost has converged for a sufficiently small ϵ. The search algorithm is expected to ignore a state once it has been labelled as solved, and to confine its exploration to the remaining unsolved states.

For an action to be selected, we require that its preconditions are satisfied by the model, and that its start effects are consistent with the other actions that are to be started at the same time. We consider an inconsistency to arise if: (1) a start effect of one action deletes a precondition of another, or (2) both positive and negative truth values are asserted for the same proposition by different start effects. As it is possible for a probabilistic event to occur at the start of an action, we restrict these rules to apply only to start effects that occur irrespective of the outcome.

The selection rules ensure that preconditions are honoured and that a degree of resource exclusion is maintained, but they do not consider other types of conditions or non-start effects. We contend that this is actually preferable, as with probabilistic outcomes we may not even know whether or not an inconsistency will actually arise. We believe that it is up to the planner to determine whether or not the risk of creating an inconsistency is worth it. For this purpose, there is an inconsistency when: (1) an asserted condition is not satisfied, or (2) both positive and negative truth values are asserted for the same proposition in the same time step. When such an inconsistency is detected, we consider the current state to be a *failure state*; a solved state with a cost of 1.

To explore this search space, we use a search algorithm that combines a deterministic search with the convergence and labelling optimisations used by LRTDP (Bonet & Geffner 2003). As with recent probabilistic temporal planners (Aberdeen, Thiébaux, & Zhang 2004; Mausam & Weld 2005), this algorithm is set in a trial-based framework, and explores the search space by performing repeated depth-first probes starting from the initial state. Unsolved states are selected to minimise $P(s)\,U(s)$, where $P(s)\,L(s)$ is used to break ties. The probability weights focus the search on the most likely states first. The lower bound guides the selection of choice states until at least one way of reaching the goal is proved to exist; the precedence of cost upper bounds causes known solutions to be robustified. The search is constrained by a finite horizon: any state with a time greater than a specified limit is a failure state. This is necessary to ensure termination. If all heuristics are admissible and $\epsilon = 0$ is used for convergence, then only optimal solutions are produced.

The search space that we have described is acyclic. Our search algorithm is general enough to work with a cyclic search space, which might arise when states are disassociated from the absolute timeline, as in (Mausam & Weld 2005).

Heuristics

Due to the added complexity from combining probabilistic effects and durative actions, effective heuristics are even more critical for probabilistic temporal planning than for simpler planning problems. A popular technique for generating cost heuristics is to use a derivative of the planning graph data structure (Blum & Furst 1997). This has been previously used for both probabilistic (Blum & Langford 1999) and temporal planning (Smith & Weld 1999; Do & Kambhampati 2002), but not for the combination of the two. We extend the planning graph for probabilistic temporal planning and use it to compute an initial *lower bound* estimate for the cost of each newly created state.

The traditional planning graph consists of alternate levels of proposition and action nodes, with edges linking nodes in adjacent levels. We also include levels of *outcome* nodes, where each node directly corresponds to a node in an action's decision tree. With this addition, edges link the nodes: proposition to action, action to outcome, outcome to outcome, and outcome to proposition. To cope with the temporal properties, we associate each level with the time step that it represents. Excepting only persistence actions, we also break the assumption that edges can at most link nodes in adjacent levels; all edges involving outcome nodes link levels of the appropriate times.

Generating a planning graph requires a state in order to determine which proposition nodes to include in the initial level. We only generate a graph for the initial state of the problem, although generating additional graphs for other states can improve the cost estimates. The graph expansion continues until the search horizon is reached. This is essential for this heuristic to be admissible, as we need to account for all possible contingencies.

Once the graph is generated, we then assign a vector of costs to each of the graph's nodes. Each component of these vectors is associated with a goal proposition; the value of a particular component reflects the node's ability to contribute to the achievement of the respective goal proposition within the search horizon.[1] In accordance with our cost usage, a value of 0 means that the node (possibly in combination with others) is able to make the goal proposition inevitable, while a value of 1 means that the node is definitely irrelevant. Cost vectors are first assigned to the nodes in the graph's final level; goal propositions have a value of 0 for their own cost component, and 1 for the others. All other propositions have a value of 1 for all cost components.

Component values are then propagated backwards through the graph in such a way that each value is a lower bound on the actual cost. The specific formulae for cost propagation are:[2]

$$C_o(n,i) \;:=\; \prod_{n' \in S(n)} C_{p,o}(n',i), \tag{5}$$

$$C_a(n,i) \;:=\; \sum_{n' \in S(n)} P(n')\, C_o(n',i), \tag{6}$$

$$C_p(n,i) \;:=\; \prod_{n' \in S(n)} C_a(n',i), \tag{7}$$

where C is the i'th cost component of node n, S are the successors of n, and P is the probability of n. Subscripts are given to C according to node type: o for outcome, a for action and p for proposition. Both C_o and C_p are admissible, and C_a is an exact computation of cost. The products in (5) and (7) are required because it might be possible to concurrently achieve the same result through different means. For example, there is a greater chance of winning a lottery with multiple tickets, rather than just the one. When a planning domain does not allow this form of concurrency, then we can strengthen the propagation formulae without sacrificing admissibility by replacing each product with a $\min$. This effectively leaves us with what has been called *max-propagation* (Do & Kambhampati 2002), which is admissible for temporal planning. Admissibility in the general case is lost when probabilistic effects are combined with concurrency.

We now explain how the cost vectors are used when computing the lower bound cost estimate for a state generated by

[1]The cost vectors can be extended to include other components, such as for resource usage or makespan.

[2]These formulae assume that every action has at least one precondition; a fake proposition should be used as a dummy precondition if this is not the case.

the search. This computation involves determining the nodes in the graph that are *relevant* to the state, and then combining their cost vectors to produce the actual cost estimate. When identifying relevant nodes, we need to account for both the state's model and event queue. Accounting for the model is not as simple as taking the corresponding proposition nodes for the current time step. Although this would be admissible, the resulting cost estimates would not help to guide the search; the same estimate would be generated for successive selection states, with nothing to distinguish between them. The way that we actually account for the model is to treat as relevant: (1) the nodes from the *next* time step that correspond to current propositions,[3] (2) the nodes for the startable actions that the search algorithm has not already considered for the current time step, if the state is a choice state, and (3) the outcome nodes for the current unprocessed probabilistic events if the state is a chance state. Those proposition or outcome nodes that are associated with an event from the event queue are also relevant.

The first step in combining the cost vectors of the relevant nodes is to aggregate each component individually. That is, to multiply — or minimise, if 'max'-propagation is being used — the values for each of the goal propositions to produce a single vector of component values. The actual cost estimate is the maximum value in this vector; the value associated with the 'hardest' goal proposition to achieve.

Planning graphs usually include mutexes, to represent binary mutual exclusion relationships between the different nodes. For our modified planning graph, we compute mutexes for all of proposition, action and outcome nodes. Of note, all of the usual mutex conditions are accounted for in some way, and there is a special rule for mutexes between action nodes in different levels to account for the temporal dimension. The complete set of mutex rules is described in (Little 2004).

Experimental Results

Prottle is implemented in Common Lisp, and is compiled using CMUCL version 19a. These experiments were run on a machine with a 3 GHz Intel processor and 900 MB of RAM. We show experimental results for a problem in a customised domain that we call maze. For this problem we vary ϵ and the use of the planning graph heuristic, while recording execution time, solution cost, and the number of states expanded; **time1**, **cost1** and **states1** are for the case where the heuristic is *not* used, and **time2**, **cost2** and **states2** are for when it is. All times are given in seconds. Recall that the costs are probabilities of failure.

The maze domain is based on the idea of moving between connected rooms, and finding the keys needed to unlock closed doors. Each of the domain's actions has a duration of 1 or 2, and many of their effects include nested probabilistic events. We used a problem with 165 action instances, although with a high degree of mutual exclusion. The tests all have a horizon of 10. The results are shown in Table 1.

We have shown that the planning graph heuristic is effective, and can dramatically reduce the number of states that are explored. As usual, the degree of effectiveness depends on the actual problem being solved.

[3]Or the current time step if it is also the last.

horizon	ϵ	time1	time2	cost1	cost2	states1	states2
10	0.0	223	11	0.178	0.178	1,374,541	13,037
10	0.1	218	3	0.193	0.178	1,246,159	2,419
10	0.2	73	1	0.197	0.193	436,876	669
10	0.3	71	2	0.202	0.193	414,414	1,812

Table 1: The maze results.

Future Work

Our work on probabilistic temporal planning is in some ways orthogonal to that described in (Mausam & Weld 2004; 2005); we believe that some of the techniques and heuristics that Mausam and Weld describe, such as combo-elimination, eager effects, and hybridization could be adapted for Prottle's framework.

At the moment, the Prottle's main bottleneck is memory usage. One way this could be improved is to compress the state space as it gets expanded, by combining states of like type and time.

There are many ways in which this framework could be made more expressive. The most important practical extensions would be to add support for metric resources, and to generalise costs to support makespan and other metrics. We understand how to do this, but have not yet implemented it.

References

Aberdeen, D., Thiébaux, S., and Zhang, L. 2004. Decision-theoretic military operations planning. In *Proc. ICAPS.*

Bacchus, F., and Ady, M. 2001. Planning with resources and concurrency: A forward chaining approach. In *Proc. IJCAI.*

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Blum, A., and Langford, J. 1999. Probabilistic planning in the Graphplan framework. In *Proc. ECP.*

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

Do, M., and Kambhampati, S. 2002. Planning graph-based heuristics for cost-sensitive temporal planning. In *Proc. AIPS.*

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Little, I. 2004. Probabilistic temporal planning. Honours Thesis, Department of Computer Science, The Australian National University.

Mausam, and Weld, D. 2004. Solving concurrent markov decision processes. In *Proc. AAAI.*

Mausam, and Weld, D. 2005. Concurrent probabilistic temporal planning. In *Proc. ICAPS.*

Smith, D., and Weld, D. 1999. Temporal planning with mutual exclusion reasoning. In *Proc. IJCAI.*

Younes, H. L. S., and Littman, M. 2004. PPDDL1.0: The Language for the Probabilistic Part of IPC-4. In *Proc. International Planning Competition.*

Younes, H. L. S., and Simmons, R. G. 2004. Policy generation for continuous-time stochastic domains with concurrency. In *Proc. ICAPS.*

Modeling Local Search: A First Step Toward Understanding Hill-climbing Search in Oversubscribed Scheduling*

Mark Roberts and **Adele Howe** and **L. Darrell Whitley**
Computer Science Dept., Colorado State University
Fort Collins, Colorado 80523
mroberts,howe,whitley@cs.colostate.edu
http://www.cs.colostate.edu/sched

Abstract

Previous results for a real-world domain, scheduling communications requests for the Air Force Satellite Control Network (AFSCN), suggested that the best search methods perform a greedy random walk. We examine the degree to which this observation is accurate for AFSCN by modeling a next-descent hill-climbing search as a Markov process. We find that the model is somewhat accurate for AFSCN. To generalize, we apply this model to another oversubscribed scheduling problem: scheduling image requests for a set of Earth Observing Satellites (EOS). We find that the hill-climber follows a trajectory that can be modeled as a Markov process.

Hill-Climbing Search as a Markovian Process

A Genetic Algorithm (GA), Squeaky Wheel Optimization (SWO), and a simple Hill-Climber (HC) have been shown to perform well for AFSCN scheduling (Barbulescu, Whitley, & Howe 2004). Simulated Annealing (SA) and HC perform best for EOS scheduling (Globus *et al.* 2004). All these methods use a permutation representation with some combination of stochastic motion and simultaneous moves. The GA selects approximately half the tasks for movement and is at the far end of multi-move, disruptive motion. The HC is at the other end of single-move, step-wise motion. SA follows a similar path as the HC, but relaxes the greedy bias by allowing uphill moves; its behavior approaches that of the HC as search progresses. SWO selects a subset of 'troublesome' tasks to move and is between the extreme moves of the GA and the small steps of the HC.

Researchers for AFSCN and EOS identify the under-performance of heuristic search in these domains, but it isn't clear exactly why this is the case. A simple hypothesis is that these methods lack a suitable ordering heuristic. Underlying this hypothesis is an implicit belief that the better performing algorithms are finding and exploiting rich, complex problem structure. Previous studies have shown that simple patterns do not account for good AFSCN schedules (Barbulescu, Whitley, & Howe 2004).

Recently for AFSCN, we have also found that an uninformed, completely random neighborhood performs better than systematic or problem specific neighborhoods (Roberts *et al.* 2005). AFSCN problems are dominated by neighborhoods with equivalent moves (plateaus and regions with many equal and only a few non-equal moves) (Barbulescu, Whitley, & Howe 2004). Based on the neighborhood result and evidence of plateaus, it seems reasonable to consider the possibility that the best algorithms may be performing a greedy random walk. Recent work in understanding the behavior of Tabu search for the Job-Shop Scheduling problem links search space features with the cost of locating an optimal solution; search can be modeled with 95% accuracy as a random walk (Watson, Whitley, & Howe 2004).

Our main focus in this work is to construct a Markov model for the simple, but reasonably successful, stochastic HC in two domains and to assess its accuracy. This is a first step toward bringing together a unifying model explaining the behavior of all of these search techniques, and possibly yielding more clues linking problem structure and run-time behavior.

Oversubscribed Scheduling

In oversubscribed scheduling, more requests need to be scheduled than can be feasibly accommodated given the available resources. This necessitates discarding some requests. We examine two oversubscribed scheduling applications: AFSCN and EOS.

For AFSCN, communication requests for earth orbiting satellites are scheduled on a set of 16 antennas at nine ground-based tracking stations. Satellites are grouped according to two orbits. Low-altitude orbits have a short visibility window and few scheduling alternatives. Typically, only one contact request can be scheduled during one 15 minute visibility window. In contrast, requests for high-altitude orbits have longer durations (20 minutes or more) and have much larger visibility windows. The scheduling alternatives can include multiple ground tracking stations. The requests to be scheduled include information about which ground stations and times are possible alternatives.

*This research was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-03-1-0233. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Customers submit requests that are scheduled by humans in a complex arbitration process. Although AFSCN starts as an oversubscribed scheduling problem, all jobs are eventually scheduled through negotiating relaxed task requirements. Our automated scheduler reduces human effort by minimizing the total number of tasks in conflict. When minimizing the number of conflicts, if the request cannot be scheduled on any of the alternative resources, it is dropped from the schedule (i.e., bumped).

Our AFSCN dataset consists of 12 days of real data identified by their dates[1]. Table 1 (left) shows characteristics for these problem instances. The seven older days of data are smaller problems that are easily solved by most of the approaches we have tried. The five newer days are substantially larger problems that are more difficult.

For EOS, as in (Globus *et al.* 2004), image capture requests are scheduled on a set of 1 to 3 identical satellites that are spaced ten minutes apart in a sun-synchronous, earth orbit at 800 km. The imaging sensor is mounted with a total possible cross-track slew of 24 degrees off nadir (straight down). A Solid State Recorder (SSR) stores on-board data and is dumped as the satellite passes over a remote tracking station located in Anchorage, Alaska. For each satellite, 2100 land-based image targets are randomly selected from the Geographic Nameserver Database (NGIA 2004). Image exposure duration ranges from 24 to 48 seconds depending on the problem. We use the free version of STK (AGI 2004) to model one week of orbits and target visibilities[2]. Not all tasks are visible from the satellites in every problem, but the number of requests ensure an oversubscribed problem.

The multi-objective evaluation function is:

$$F = w_p \Sigma_{u \in U} P_u + w_s \mu_{slew} + w_a \mu_{angle}$$

where w_p, w_s, w_a are the weights assigned in the problem definition and where μ_{angle}, μ_{slew}, and $\Sigma_{u \in U} P_u$ are the average image angle, the average slew per task, and the sum of the unscheduled image request priorities, respectively. For this paper, we round the value of F to the nearest integer.

Our EOS dataset consists of ten problem instances using problem parameters outlined in (Globus *et al.* 2004). These problems are specifically designed to mimic realistic satellite scheduling problems. Table 1 (right) shows characteristics for these problem instances. The last three columns show the weights for the evaluation function.

A Simple Hill-climbing Algorithm

We encode potential solutions using a permutation π of the n task IDs, $[1..n]$. A *schedule builder* is used to generate solutions from the permutation. In effect, the permutation π acts as a priority queue, and the schedule builder places task requests in the schedule based on the order that they appear in π. Each task request is assigned to the first available resource from its list of alternatives and at the earliest possible starting time. This assignment treats the list of alternatives as a rank order, although the actual ordering is arbitrary.

We implemented a next-descent HC that employs the *shift* operator; we accept new solutions that are better or equally good. From a current solution π, a neighborhood is defined by considering all $(n-1)^2$ pairs (x, y) of positions in π, subject to the restriction that $y \neq x - 1$. The neighbor π' corresponding to the position pair (x, y) is produced by *shifting* the job at position x into position y, while leaving all other relative job orders unchanged. If $x < y$, then $\pi' = (\pi(1), ..., \pi(x-1), \pi(x+1), ..., \pi(y), \pi(x), \pi(y+1), ..., \pi(n))$. If $x > y$, then $\pi' = (\pi(1), ..., \pi(y-1), \pi(x), \pi(y), ..., \pi(x-1), \pi(x+1), ..., \pi(n))$.

Building a Model

Our goal is to model the expected cost of moving from a random solution to a solution with the best known value. By expected cost, we mean the average number of shifts made in the shift neighborhood. To estimate the search cost, we construct a Markov model of search progress from a set of independent runs and calculate the waiting time from each state. In using a Markov model we make two assumptions: 1) that movement to the next state is entirely dependent on the current state and 2) that we can make a suitable estimate of the transition probabilities between states.

We adapt the methodology from (Frank, Cheeseman, & Stutz 1997) to name states and 'probabilistically paint' the schedule space. For a given problem instance $\mathcal{I}$ with n tasks, let G be an undirected graph of the $n!$ permutations. Two vertices of the graph, g_1, g_2 are adjacent, that is, have an edge between them, if they correspond to two permutations differing by a single shift. G is the search space induced by the shift operator. We associate G with the evaluation (schedule) space by assigning each vertex the objective function value from the schedule builder $\mathcal{B}$ for $\mathcal{I}$.

Definition 1 ***(Level)*** *Let $f : G \rightarrow \mathcal{Z}^+$ be a function mapping permutations to integers such that $f(\mathcal{B}(g)) = z$ if and only if the permutation corresponding to g results in an evaluation of z using schedule builder $\mathcal{B}$.*

Definition 2 ***(Plateau)*** *Let P be a connected subgraph of G and let $z \in \mathcal{Z}^+$ be a constant. Then P is a plateau if P is a maximal connected subgraph of G such that $f(\mathcal{B}(p)) = z$ for all $p \in P$. Further, z is defined to be the level of the plateau.*

Definition 3 ***(Aggregate Plateau)*** *Let A be the set of all plateaus in G at level $z \in \mathcal{Z}^+$. Then A is an aggregate plateau if every plateau $a \in A$ is a plateau at level z and every plateau $p \in G$ at level z is in the set A. Further, z is defined to be the level of the aggregate plateau.*

Definition 4 ***(Model State)*** *Let s be a single state in the Markov model with a set of states $\mathcal{S}$ and a transition probability matrix $\mathcal{T}$. Then s is an aggregate plateau in G at level $z \in Z^+$. Further, s_z is defined to be the level of the state. A transition probability, t_{ij}, signifies the probability of moving from state s_i to s_j. Let s_{min} to denote the lowest s_z in the model and s_{max} to denote the highest s_z in the model; $0 < s_{min} < s_{max} < O(n)$.*

[1]We thank Dr. James T. Moore, Associate Professor, Dept. of Operational Sciences, Air Force Institute of Technology and Brian Bayless and William Szary from Schriever Air Force Base for providing the AFSCN data.

[2]We thank Al Globus of CSC at NASA Ames for walking us through retrieving target visibilities from STK.

ID	Date	Size	# Low	# High
Day1	10/12/92	322	153	169
Day2	10/13/92	302	137	165
Day3	10/14/92	311	146	165
Day4	10/15/92	318	142	176
Day5	10/16/92	305	142	163
Day6	10/17/92	299	144	155
Day7	10/18/92	297	142	155
Mar07	03/07/02	483	225	258
Mar20	03/20/02	457	194	263
Mar26	03/26/03	426	183	243
Apr02	04/02/03	431	185	246
May02	05/02/03	419	178	241

ID	Size	SSR	Slew	SSRUse	Priority	w_p	w_s	w_a
1	1 (2100)	75	2	1	1-6(50)	1	0.01	0.02
2	3 (6300)	50	2	1,3,5	1-6(50)	1	0.01	0.50
3	3 (6300)	50	2	1,3,5	1-6(50)	1	0.01	0.00137
4	3 (6300)	75	2	1,3,5	1-6(50)	1	0.01	0.02
5	3 (6300)	50	10	1,3,5	1-51(5)	1	0.50	0.20
6	3 (6300)	75	2	1,5,8	1-16(50)	1	0.10	0.20
7	3 (6300)	75	2	1,5,8	1-16(50)	1	0.10	0.20
8	3 (6300)	75	2	1,3,5	1-6(50)	1	0.01	0.02
9	2 (4200)	75	2	1,3	1-6(50)	1	0.01	0.02
10	3 (6300)	100	1	1,10,25	1-6(50)	1	0.10	0.70

Table 1: The left table shows AFSCN problem characteristics for the 12 days of data used in our experiments. *ID* is used to identify the instance throughout the paper. *Size* is the number of requests in the problem. *# Low* and *# High* are the number of low and high-altitude requests in each problem. The right table shows the problem characteristics for the EOS problem. *ID* is used to identify the instance throughout the paper. *Size* is the number of satellites (image requests) in the problem. *SSR* is the size of the on-board storage. *Slew* is the speed of the slew motor in degrees/second. *SSRUse* is the amount of SSR usage and is uniformly divided among the tasks in the problem. *Priority* is the value range (number of levels) for priority assignments to tasks. The last three columns signify the weights for the evaluation function.

To summarize, we aggregate states into sets partitioned by level. An aggregate plateau combines all states of the same level regardless of whether they are actually on the same plateau. It is intractable to explicitly model every solution ($\Theta(2^n)$ for the schedule space or $O(n!)$ for the permutation space). Using the level as a surrogate state makes many assumptions about the independence between schedules and movement between states. These assumptions could lead to model inaccuracy.

The size of the AFSCN model is straightforward: s_{min} denotes the best value seen and s_{max} denotes the highest value seen during the search. We are reasonably sure that s_{min} is in fact the optimal value for each problem.

Memory and time limitations prohibit us from constructing a full EOS model. We are also less confident that we've located the best values for EOS. To capture the runtime dynamics for the 'settled' behavior of the HC, we model the lowest and largest portion of search that we can. EOS runs incur a high computational cost. To include as many runs as possible, we set s_{min} to $max(finalEval(r)), r \in R$, where R is the set of runs used to construct the model. We limit the model size such that $s_{max} = s_{min} + 1000$. These model choices assume that all runs are equal, which could lead to a negative result if some runs are stuck on suboptimal plateaus.

We run each problem for as many evaluations as we can given our computing resources. For AFSCN, we stop the search at 50,000 evaluations per run (2 to 10 minutes of CPU time). For EOS, we stop search at 100,000 evaluations per run (almost one day of CPU time).

We estimate the transition matrix by collecting transition counts from a set of independent runs. For AFSCN, we use 25 runs per state; each run is guaranteed to reach s_{min}. For EOS, we use approximately 50 runs for the entire model. From these transition counts, we calculate the transition probabilities for the states. We then use the transition matrix to calculate the expected waiting time from each s to s_{min}. In our analysis of the model, we make two simplifying assumptions: s_{min} is absorbing, and s_{max} is a reflecting barrier.

```
VALIDATE-ENTRY
1   $numShifts \leftarrow 0$
2   while $v$ in $S_{validate}$ has less than $m$ runs
3       do repeat $x \leftarrow$ random solution
4             until LEVEL$(x) > v_z$
5          while LEVEL$(x) > v_z$
6              do HC proceeds one shift
7          while LEVEL$(x) > s_{min}$
8              do HC proceeds one shift
9                 $numShifts \leftarrow numShifts + 1$
10  return $numShifts$
```

Figure 1: Pseudo-code for VALIDATE-ENTRY. LEVEL(x) returns the integer level of the solution x using the appropriate schedule builder. All runs reach s_{min}.

Validating The Model

We are interested in measuring how well the model predicts the search cost from each model state given as large as possible a set of independent runs. For each state in the model, we correlate the predicted cost of the model with the actual cost of arriving at s_{min}. We obtain m validating runs for each state in the set of states we wish to validate, $\mathcal{S}_{validate}$. Figure 1 shows the algorithm for obtaining the actual search cost of each state $v \in \mathcal{S}_{validate}$. We observed that the run length distributions for most states $\mathcal{S}_{validate}$ have heavy tails, so we correlate the model cost with the *median* actual cost.

For AFSCN, we show results for all 12 days of data where $m = 25$ and $\mathcal{S}_{validate} = \mathcal{S} - \{s_{min}\}$. For EOS, we present a random sampling of the ten problems where $m = 7$ and

AFSCN	r^2
Day1	.040
Day2	.855
Day3	.743
Day4	.643
Day5	.052
Day6	.716

AFSCN	r^2
Day7	.844
Mar07	.569
Mar20	.611
Mar26	.287
Apr02	.720
May02	.688

EOS	r^2
1	.805
3	.851
6	.805
7	.839
9	.831
10	.606

Table 2: R-squared values for the hypothesis that the correlation for a given problem is linear.

$\mathcal{S}_{validate} = \{s_{min+1}, s_{min+333}, s_{min+666}, s_{max}\}, s \in \mathcal{S}$. Table 2 shows the r-squared values for linear regression. This preliminary analysis shows that there is a strong linear relationship between the model for most of the problems.

Summary and Future Work

For two oversubscribed scheduling problems, we hypothesize that a random walk model may capture the dynamics of search. We built a Markov model of the search using aggregate plateaus as the states. We found that this model is well-correlated for EOS, though it is less consistently correlated for AFSCN. This is simply the first step toward explaining search in such problems. We now identify some possible sources of model failure that we are currently addressing.

First, we note that states far away from s_{min} are much less correlated with the actual search cost. This effect is most noticeable for the larger problems in AFSCN. Figure 2 shows the correlation plot for Mar07, where this compression of states is seen on the right side. We conjecture that this compression results from a lack of sufficient state for the large plateaus seen during search. So we are currently building a more complex model with additional states representing the length of a plateau walk. This model refinement may also provide a more clear estimate of the plateau size for these problems. Preliminary evidence suggests that the larger model does resolve some of this compression.

Second, the model is clearly lacking in strength for some problem instances. Specifically, Day1, Day5, Mar26, and EOS10 show low correlation in comparison to the other problem instances. There is strong evidence that the compression mentioned above plays a role in the low correlation. The random walk may not be the best model for these problems; in which case, we will seek to capture the differences between EOS and AFSCN that might explain the differences in performance.

Third, the issue of knowing the best-value for EOS is still open, and search could be made more efficient. We will continue our analysis of the model as other problems accumulate more runs.

Finally, we have only taken one step toward modeling the simplest algorithm for these domains. We hope to generalize this model to other algorithms with the goal of finding a unifying model.

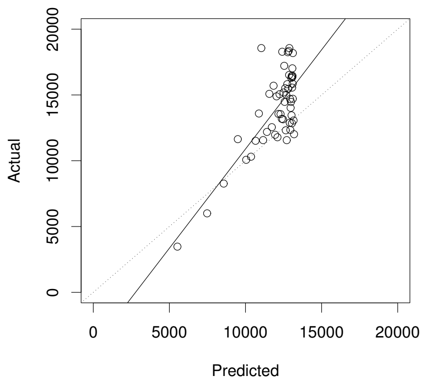


Figure 2: The correlation plot for Mar07, the largest AFSCN instance. The dashed line indicates perfect correlation. This plot reveals evidence of compression in the model for states far away from the optimum.

References

(AGI) Analytical Graphics, Inc. Satellite Toolkit (STK). www.stk.com. Accessed Sept. 9, 2004.

Anderson, E. J. 2002. Markov chain modeling of the solution surface in local search. *Journal of the Operational Research Society* 53:630–636.

Barbulescu, L.; Whitley, L.; and Howe, A. E. 2004. Leap before you look: An effective strategy in an oversubscribed problem. In *Proceedings of the Nineteenth National Artificial Intelligence Conference (AAAI-04).*

Frank, J.; Cheeseman, P.; and Stutz, J. 1997. When gravity fails: Local search topology. *Journal of Artificial Intelligence Research* 7:249–281.

Globus, A.; Crawford, J.; Lohn, J. D.; and Pryor., A. 2004. A comparison of techniques for scheduling earth observing satellites. In *Proceedings of the Sixteenth Conference on Innovative Applications of Artificial Intelligence*, 836–843.

Kemeny, J. G., and Snell, J. L. 1976. *Finite Markov Chains.* New York: Springer-Verlag. chapter Absorbing Markov Chains, 43–68.

(NGIA) National Geospatial-Intelligence Agency. Geographic Name Server (GNS). http://gnswww.nima.mil/geonames/GNS/index.jsp. Accessed Aug. 31, 2004.

Roberts, M.; Whitley, L. D.; Howe, A. E.; and Barbulescu, L. 2005. Random walks and neighborhood bias in oversubscribed scheduling. In *2nd Multidisiplinary Conference on Scheduling: Theory and Applications (MISTA-05)*, to appear.

Watson, J.; Whitley, L.; and Howe, A. 2004. Linking search structure, run-time dynamics, and problem difficulty: A step toward demystifying tabu search. *Journal of Artificial Intelligence Research* under submission.

Planning on demand in BDI systems

Lavindra de Silva and Lin Padgham
{*ldesilva,linpa*}*@cs.rmit.edu.au*
School of Computer Science and Information Technology
RMIT University, Melbourne 3000, VIC, Australia

1 Introduction

The BDI (Belief, Desire, Intention) model of agency is a popular architecture based on Bratman's (Bratman 1987) theory of practical reasoning. There are numerous implementations based on the BDI architecture such as PRS (Georgeff & Ingrand 1989) and JACK[1], which are used for both academic and industrial purposes. An important aspect of BDI style systems is that they execute as they reason, and so avoid the possibility of the reasoning being outdated, due to environmental change, by the time execution happens. This makes them useful for many complex and dynamic environments, such as Unmanned Autonomous Vehicles (UAVs) and Air Traffic Management, due to their ability to cope well with changes, making adjustments as they go in terms of the steps chosen. They are also very fast, and therefore well suited to systems needing to operate in real time, or close to real time environments. However, there are no generic mechanisms in BDI systems to do any kind of look-ahead, or planning. In some situations this would be desirable.

The primary goals and contributions of our work are: 1) incorporating planning at specific points in a BDI application, on an as needed basis, under control of the programmer; 2) planning using only limited subsets of the application, making the planning more efficient, and; 3) incorporating the plan generated back into the BDI system, for regular BDI execution, identifying plan steps that could be pursued in parallel.

Other features of our approach include: 1) minimising the programming overhead, as the program to be run by the planner is derived from the existing BDI program; 2) allowing the use of regular functions in planning; 3) extracting planning effects from existing code, and; 4) incorporating aspects of both HTN (Hierarchical Task Networks) planning and classical planning.

There is some previous work that deals with using planning capabilities to guide the execution of BDI-like systems. Some of the research closely related to ours is Propel (Levinson 1995), Propice-Plan (Despouys & Ingrand 1999), and RETSINA (Paolucci *et al.* 1999). Propice-Plan (and other similar systems) use a reactive system for execution, and a separate planning component to produce plans. There are three important differences between our system and Propice-plan : 1) the planner provides an alternative means of operation; it is not smoothly integrated into the BDI architecture and algorithms. In particular, the planner does not provide guidance to the reactive-planner, on how to use its hierarchy to achieve the goal, instead, the reactive planner only executes the operators in the final plan produced; 2) Propice-plan does not allow the programmer to specify points when the planner should be called, and to provide extra operators for each planning point; 3) Propice-plan does not support the use of programmer provided functions in planning, i.e. where the outcome cannot be specified as an effect. Propel and RETSINA are different to Propice-plan in that planning and execution are tightly integrated within a single framework. However, in Propel, the planner is called every time a runtime failure occurs, whereas our system allows the programmer to specify when to call the planner, and what part of the hierarchy to use for planning. In RETSINA, planning is done when all the information required for planning is available, therefore not leaving the decision of when to plan and when to execute to the programmer.

A simplified view of the BDI architecture is in terms of *goals* and *recipes*, where a goal has one or more recipes that can be used to achieve it. The recipes to achieve goals are stored in a library provided by the programmer. When an agent has a goal to achieve, it looks for a recipe that can achieve the goal in the current state of the world. If a suitable recipe is found, it is executed. If it fails during execution, the agent looks for other suitable recipes to achieve the goal. The goal fails if none of its recipes could be executed to completion, or if none of them were suitable for the current state of the world.

In achieving a goal, the agent typically executes a number of steps, or *subgoals/subtasks*. In some situations there can be multiple options (recipes) at each step, but for a given state, only certain combinations of choices will lead to success of the overall goal. However, it may not be possible to (easily) encode information enabling successful choices, based only on knowledge of the current state. If steps are cheap, reversible, and there is little or no cost to doing them, then a BDI system can easily be used to find a solution in these situations. However, in many cases of this sort in real applications, actually doing the actions rather than only rea-

[1]JACK Intelligent Agents : http://www.agent-software.com.au

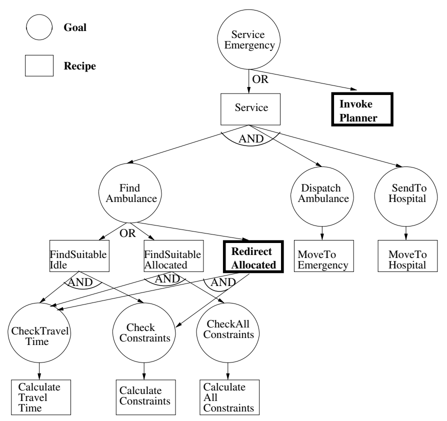


Figure 1: The Design of the BDI Program. Additional planner recipes are highlighted in bold.

soning about them (planning), results in wasting time and resources in ineffectual behaviour. At times this can even lead to inability to achieve the goal. While it is always possible for planning to be explicitly programmed into a BDI application (and in fact this is often done), it would be advantageous to have a simple mechanism that incorporates planning in a generic way, at particular points where it is needed.

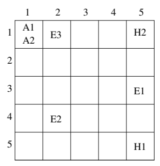
2 Example

Let us present an example to illustrate the inclusion of a planning facility within a BDI application. Consider an ambulance dispatch system: as ambulance requests arrive, a dispatching agent ascertains the best ambulance to dispatch and provides ongoing directions during the process of dealing with the emergency.

The steps that are part of servicing the emergency are *FindAmbulance*, *DispatchAmbulance* and *SendToHospital*, in that order, as shown in Figure 1 (ignore recipes in bold for now). Normally the FindAmbulance goal is achieved by executing the *FindSuitableIdle* recipe. This is a recipe type representing a set of recipe instances. These instances are determined at runtime, one for each idle ambulance. Each of these recipe instances will be tried in turn, attempting to successfully complete the subtasks of *CheckTravelTime* and *CheckConstraints*. If no recipe instance of this type completes successfully, instances of the alternative recipe type *FindSuitableAllocated* will be tried. This recipe attempts to schedule the new emergency to follow one of the jobs that is already allocated.

However, if all instances of all applicable plan types fail, i.e. are unable to be done in a way that meets the time constraints for the new emergency, the *ServiceEmergency* goal will fail. Since at this point the BDI system cannot find a solution, it could be useful to call a planner to see if there is a way to successfully achieve this *ServiceEmergency* goal, while also servicing all other such goals and maintaining all timing constraints. The planner could attempt to develop a plan that reallocates some of the currently allocated emergencies, so that the new emergency can also be serviced.

For example, the situation may be that there are two ambulances and two hospitals, placed on the grid as shown in Figure 2. Three emergencies occur at different times, at locations also shown on the grid. The table beside the grid



Time	Emergency	Urgency
0	E1	10
2	E2	18
5	E3	15

Figure 2: Ambulance servicing example

gives the number of time units after the start that each emergency happens and the urgency of the emergency. Urgency is in terms of maximum time units acceptable between the emergency happening, and the ambulance having serviced the emergency arriving at the hospital. We assume that the ambulances can move right, left, up and down, and that each move takes one time unit.

Using the BDI recipes in Figure 1, the first two emergencies will be allocated to the two ambulances. Assume E1 is allocated to ambulance A1, and to hospital H1, while E2 is allocated to ambulance A2, and also to hospital H1. It will take eight time units to service each of E1 and E2.

When emergency E3 occurs, the ambulances are busy, so the recipe *FindSuitableAllocated* will be tried, to see if E3 can be scheduled with either A1 or A2, after their current job, in a way that meets the urgency constraints. In this case A1 and A2 arrive at H1 at time units 8 and 10 respectively (from the start time). The trip from H1, to E3, and then to H2 takes 10 units. As it is necessary to service E3 by 20 time units from the start (occurrence time plus urgency allowance) it is possible to allocate E3 to be done by A1 after it has reached H1. This will have E3 at the hospital at time unit 18, 2 time units before the maximum for its urgency level.

However, if E3 had had an urgency level of 10 units, then this allocation would not have been successful. By calling a planner, and exploring possible re-allocations, it would be possible to allocate A1 to service E1 and E2, while A2 services E3. To arrive at this set of allocations the planner must find an appropriate set of bindings to meet all constraints for the three *ServiceEmergency* goals. I.e. find a set of bindings for the first goal, project it's effects, then find a set of bindings for the next goal based on the projected state, and continue in this manner. If a set of bindings could not be found for a goal, an alternative set of bindings would need to be found for the previous goal.

3 Overview

What we propose in this work is a mechanism whereby a BDI programmer can indicate that runtime planning should be applied. This may be on failure of a more simple approach (as in our example), or in a situation where planning may always be appropriate for a particular task or subtask. Our approach is to use the information already available within the BDI program as the knowledge which must be provided to the planner, and to thereby relieve the programmer of the responsibility of specifying information specifically for planning. The planner can then make choices about which recipe instances to use and how these are sequenced.

In order to provide the required information to the planner, the relevant BDI goals and/or recipes, as well as the relevant information about the current state as captured by agent beliefs, must be translated into a representation suitable for some planning system. Due to the similarity of problem representation between BDI systems and HTNs (de Silva & Padgham 2004), we decided to use an HTN planner which allows representation of the information encoded in the hierarchical BDI structure. We have decided to use the Java version of the SHOP planner called JSHOP[2] as it can handle numeric computations and also allows user defined Java functions to be used during planning. The BDI platform we are using is JACK, BDI agent platform, with research licensing available to universities.

The approach that we use is to have the programmer include in the BDI program a generic recipe that invokes the HTN planner at the desired point. At compile time our system then automatically converts the relevant goals and recipes[3] into a JSHOP program, which can be accessed at runtime, if the recipe invoking the planner is instantiated. After invoking the planner, the recipe executes the plan returned. If desired, additional recipes can be provided within the BDI program, for the express purpose of being available for use in planning. A suitable context condition can ensure that they are not instantiated at other times.

The process of converting information from the JACK program, into a suitable JSHOP representation, is a straightforward one-one mapping of JACK's goals and recipes (including each recipe's precondition and effect), into JSHOP's syntax (i.e. *methods*, and *precondition* and *tail* pairs). Due to space constraints, the mapping process will not be discussed here. The reader can refer to http://www.cs.rmit.edu.au/l̃desilva/research/publications/ for a more detailed paper.

The bold recipes in Figure 1 show the recipe *InvokePlanner* which calls the planner, and also recipe *RedirectAllocated* provided specifically for use by the planner.

4 Invoking the Planner

As indicated in section 3, a recipe is placed at whatever point the programmer wishes the planner to be invoked. This recipe will be chosen according to normal BDI principles. Therefore it can have a context condition which captures the situation in which it should be used, or it can be a last priority, if other options have failed (as in our example situation), or it can be prioritised with respect to alternative options in the same way that other alternative plans are prioritised (in JACK using either a precedence attribute or order of declaration).

The recipe for invoking the planner has a generic form, and therefore most of the details are added automatically, based on a template. There are four subtasks which are included within this recipe. These are to: 1) create the initial set of beliefs, corresponding to the current state when the planner is to be invoked; 2) create the desired goal state; 3) instantiate and call the planner, and; 4) execute the plan.

The list of beliefs that are relevant, and whose values should be accessed at runtime, and provided to JSHOP, must be specified by the programmer (via a GUI). (These could be extracted automatically, but this has not currently been done.) In our ambulance example, relevant beliefs are *AmbulanceState*, *EmergencyState* and *HospitalState* and values obtained at runtime include *A1* is at *4,3* and *A2* is at *2,3* from AmbulanceState, *E1* is at *5,3*, *E2* is at *2,4*, and *E3* is at *2,1* for EmergencyState and so on. When the *InvokePlanner* recipe is executed, the current values are obtained and the initial state for JSHOP is built.

The desired goal state must be provided by the programmer in the form of a recipe which can produce the desired goal state for the specific situation. For the ambulance example, the desired goal state is: *EmergencyState E1 == Serviced*, *EmergencyState E2 == Serviced*, and *EmergencyState E3 == Serviced*. This is produced using a recipe which obtains all current emergencies, including the new one, and creates a beliefset instance which has all of them as being serviced. An interface is currently being developed to support creation of most of this recipe automatically.

The third subtask is to call the appropriate JSHOP program with the initial state and goal state produced by the first two subtasks. The final task is to execute the plan which is returned. This is done using a recipe supplied by our system, and is explained in section 6.

The planner is able to do both planning by reduction within a particular goal, and also planning with multiple instances of top level goals, in a manner similar to how first principles planners plan with multiple "instances" of operators. The latter is possible because HTNs are more expressive than STRIPS style planners, as described in (Erol, Hendler, & Nau 1996). The ability to combine these techniques allows for considerable flexibility, allowing user defined functions, and a lossless representation of the BDI hierarchy.

5 Producing a Suitable Partial-Order Plan

Certain planners, including JSHOP, produce a totally ordered sequential plan. However BDI systems like JACK are intended to be able to pursue multiple goals in parallel. In order to take advantage of this aspect of BDI systems, it is desirable to parallelise the resulting JSHOP plan before beginning execution within JACK.

An alternative would be to directly use a partial-order planner. However, none of the partial-order HTN planners are Java based and would therefore not facilitate the direct integration of functions that we require. Following the algorithm described in (Veloso, Pérez, & Carbonell 1991), with minor changes, we have modified JSHOP to have a final processing step which creates a partial-order plan.

In its original form, JSHOP produces a sequence of primitive actions as the outcome of planning. These are parts of JACK recipes. We now need to execute the plan found, and we do that by incorporating it back into JACK, as there may be other aspects of the original recipes that require execution, as well as those parts that were provided to the planning process. As JACK executes recipes as a result of goals being instantiated, we need our resulting plan to be in terms

[2]JSHOP is a state of the art planner being used by the *Naval Research Laboratory for Noncombatant Evacuation Operations* http://www.cs.umd.edu/projects/shop/download.html

[3]The relevant goals and recipes are the recipes which are siblings of the planning recipe in the BDI hierarchy, and all their children recipes and goals. Other goals and recipes in the program are excluded because they are not responsible for achieving the task for which runtime planning is desired.

of goals. Consequently we have modified JSHOP to provide information on the reduction process itself; i.e. for each operator/action in the plan, which recipe and goal instance were selected to lead to that action[4].

6 Executing the JSHOP Plan

The partial-order plan returned from the planner consists of a partial order of nodes. Each node contains the top level goal, and all information necessary for binding variables and making choices of recipes as that goal is executed.

The recipe provided by our system posts the top level goals in the appropriate order. It initially posts asynchronously, all goals at the start of the plan which can be run in parallel. As each top level goal completes, any immediate successor, for which all predecesors have completed, is posted. In our example, the goal instances *SeviceEmergency E1* and *ServiceEmergency E3* are posted initially. When *ServiceEmergency E1* completes, *ServiceEmergency E2* is posted, as it is dependent only on *ServiceEmergency E1* in the partial order. If the ordering was such that *ServiceEmergency E2* was after both *ServiceEmergency E1* and *ServiceEmergency E3*, then it would not be posted until both completed.

When each goal is posted (both the top level goal and the subsequent subgoals), the BDI system must decide the appropriate recipe to use. This is based on the plan that has been returned by the planner. We require firstly that the recipe instance chosen is of the same type as that indicated by the plan. Secondly it must contain the same bindings in the context condition as that indicated in the plan.

Recipe selection is handled transparently to the programmer, by extra code added to each recipe's *context()* condition at the compilation stage. The code ensures that, when appropriate, the BDI program selects recipes based on plans returned by JSHOP, and at other times selects recipes using normal BDI recipe selection.

If at any point in the execution it is not possible to match a recipe from what JACK considers is available with what the planner considers should be executed, then this indicates that there is a problem, probably resulting from some environmental change. If at this stage execution continues, using the recipe chosen by the planner, this is likely to cause problems. JACK context conditions are written to ensure that appropriate plans for the situation are the ones that are considered. If a recipe is used which is intended for a different situation than the one existing, then it cannot be expected to succeed. If on the other hand we allow JACK to choose a recipe outside the plan which has been produced, we invalidate the plan.

In such cases, a recipe will not be selected, causing the goal it handles to fail, therefore causing the top level goal called within InvokePlanner (used as a generic term here to represent any plan that invokes JSHOP) to fail. When InvokePlanner realises the goal state has not been achieved, instead of calling the planner to replan, the InvokePlanner recipe will also fail. At this point the BDI system's failure handling will take over.

[4]Refer to http://www.cs.rmit.edu.au/l̃desilva/research/publications for more information.

7 Conclusions

BDI systems are robust in dealing with complex and dynamic environments, and work with a recipe library provided by the programmer. In some situations it can be desirable to do some planning, either as a result of other approaches failing, or in order to look ahead to guide choices at a particular point. The planner would ideally be able to use information about the existing BDI program to simulate the behaviour of the system, and provide advice on the choices the system should take during execution. We have implemented a system that does this, by using an efficient HTN planner. Our focus is different to past work in interleaving planning and execution, in that we cater for the intrinsic needs of the BDI architecture. In particular, we leave the choice of when planning should be done, and with what information, to the BDI programmer. Executing the plan is done using regular BDI execution, using the advice from the planner on what recipes to choose, and what bindings to use in context conditions. Furthermore, our plan execution model is unique, in that it is possible for the BDI system to maintain control on plan failure, and resume normal BDI execution.

We are currently working on creating formalisms to define and evaluate our framework.

8 Acknowledgements

We thank Michael Winikoff, Gaya Jayatilleke and John Thangarajah for feedback and reviews, and Agent Oriented Software, for providing the JACK Language for our work.

References

Bratman, M. E. 1987. Intention, Plans and Practical Reason, Havard University Press, Cambridge, MA, ISBN (Paperback): 1575861925.

de Silva, L. P., and Padgham, L. 2004. A Comparison of BDI Based Real-Time Reasoning and HTN Based Planning. In *17th Australian Joint Conference on Artificial Intelligence, Cairns, Australia.*

Despouys, O., and Ingrand, F. F. 1999. Propice-Plan: Toward a Unified Framework for Planning and Execution. In *European Conference on Planning (ECP)*, 278–293.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence* 18(1):69–93.

Georgeff, M., and Ingrand, F. 1989. Decision making in an embedded reasoning system. In *Proceedings of the International Joint Conference on Aritificial Intelligence (IJCAI)*, 972–978.

Levinson, R. 1995. A general programming language for unified planning and control. *Artifial Intellgence* 76(1-2):319–375.

Paolucci, M.; Shehory, O.; Sycara, K. P.; Kalp, D.; and Pannu, A. 1999. A Planning Component for RETSINA Agents. In *Agent Theories, Architectures, and Languages*, 147–161.

Veloso, M. M.; Pérez, M. A.; and Carbonell, J. G. 1991. Nonlinear Planning with Parallel Resource Allocation. In *Workshop on Innovative Approaches to Planning, Scheduling and Control*, 207–212.

Challenges in Adapting Automated Planning for Autonomic Computing

Biplav Srivastava
IBM India Research Laboratory
Block 1, IIT Delhi, Hauz Khas,
New Delhi 110016, India.
Email: sbiplav@in.ibm.com

Subbarao Kambhampati
Dept. of Computer Sc. & Engg.
Arizona State University
Tempe, AZ 85287, USA.
Email: rao@asu.edu

Abstract

Computing systems have become so complex that the IT industry recognizes the necessity of deliberative methods to make these systems self-configuring, self-healing, self-optimizing and self-protecting. Architectures for system self-management, also called Autonomic Computing (AC), have been proposed where elements are managed by monitoring and analyzing behaviors and using the response to plan and execute new actions that take or keep the system in desirable states. In this paper, we highlight some of the challenges and research problems raised in adapting automated planning techniques to AC applications.

Introduction

The vision of Autonomic Computing (AC)(Kephart & Chess 2003) is to improve manageability of complex IT systems by making them self-configuring, self-healing, self-optimizing and self-protecting. This would require that the behavior of system elements are monitored and analyzed, and the performance is used to plan and execute suitable actions to take or keep the system in desirable states.

Policy is a popular term in industry to refer to any declarative specification of behavior that is desired from a software system (e.g., agent) and the behavior is enforced by a policy engine. Two types of policies are easily distinguishable. In the first case, the policy describes desired behavior and exhaustively lists necessary actions to meet them under all conditions. During runtime, a policy engine will verify the conditions and take the stipulated action. This type of policy is procedural in nature because the actions to take under a condition is fully known, and it is suited for reactive reasoning. In the second case, the policy only lists the system's expected behavior (e.g., goal state) and it is left to the policy engine to deliberate and determine what actions need to be taken to ensure the satisfaction of goals. A generalization of goal type policy can include utility information so that the selection of actions depends on runtime situations. *Planning provides the policy engines for goal-type policies.* Planning is thus critical for meeting the AC vision.

Planning is a very wide discipline characterized by how the environment, the agent's goal and its model of the world are represented. Planning algorithms are best understood as a refinement search over sets of possible plans - an algorithm starts from the set of all possible plans and performs refinements on the plan set leading to sub-sets from which extracting a single solution is feasible(Kambhampati & Srivastava 1995; Kambhampati et al 1995). Various types of planners can return sequential(Weld 1999), conditional plans or a generalized state-action mapping (Blythe 1999) specifying what action to take in any state during execution(hence procedural policies), that are optimized with respect to a defined metric. In terms of performance, planning has seen an upsurge in the last 6-7 years with new planners that are orders of magnitude faster than before and are able to scale this performance to complex domains, e.g., metric and temporal constraints.

Despite the obvious potential for connections between automated planning and autonomic computing, very little has been done to exploit the synergy. In this paper, we highlight some of the challenges and research problems raised in adapting automated planning techniques to AC application scenarios so that researchers in the planning community become aware of the potential applications in autonomic computing. Here is the outline of the paper: we start with a brief overview of planning followed by description of AC scenarios and how planning can be useful there. We then identify AC-specific planning challenges. They include working with incomplete domain models and in managing life cycles of plans. We conclude by relating planning to procedural policies and its connections to Web and Grid services and finally give closing comments.

Preliminaries

We review planning and their role in AC scenarios in this section.

Planning

For the purposes of this paper, we introduce a planning problem PP as a 4-tuple $\langle P, I, G, A\rangle$ where P is the set of predicates, I ($\subseteq P$) is the complete description of the initial state, G ($\subseteq P$) is the partial description of the goal state, and A is the set of executable (primitive) actions. A specification of an action consists of preconditions ($A_i^{pre} \subseteq P$) and postconditions ($A_i^{post} \subseteq P$). A plan for PP is an action sequence

S, such that if S is executed in I, the resulting state of the world would contain G. A planner finds a plan by efficiently searching in the space of possible states configurations or action orderings (plans). It is desirable that a planner be *sound* and *complete*. A planner is sound if it will only generate correct plans. A planner is complete if it will always find a plan, provided one exists, given a domain and problem description. Automated planners are designed to be sound and complete.

A plan can be obtained without a planning algorithm and without explicit action specifications. An example of the former is when the user provides the plan directly and an example of the latter is when a plan is generated by some domain-dependent reasoning on initial and goal states. However, the soundness and the completeness of plan generator cannot be guaranteed. Domain-dependent planners usually produce superior plans than domain-independent methods, but they are harder to build and cannot be reused.

The user or system need not act on a plan immediately. A plan may be one of the many plans that are produced by users or planning algorithms before some plan is executed. They could be stored, searched, inspected, evaluated, modified, and critiqued by human experts or automated reasoning systems, and executed. Eventually, plans will outlast their utility and be replaced.

Planning needs of Autonomic Computing Scenarios

In the AC vision(Kephart & Chess 2003), four aspects of self-management have been identified. We discuss the role of planning in these aspects.

Self-configuration: deals with installation, configuration and integration of IT systems. The installation procedures work by gathering information about the host environment, figuring out the dependencies among needed tasks and also optimizing performance measures, and finally executing the tasks to realize the changes. Information about host system is increasingly getting standardized along structured formats but the executable tasks can be ad-hoc scripts. Humans want to be closely involved in key decisions during execution.

Self-healing: deals with determination of problematic situations and recovering from them. It requires the system to reason with how activities can be performed, how diagnostic information is produced and how new changes can be affected with minimal cost and maximum benefit. The specification of actions could be known at some level of granularity.

Self-optimizing: deals with improving the performance of running systems by leveraging alternative opportunities. The system would monitor its performance and based on its changing environment, could initiate new changes (e.g., resource re-provisioning).

Self-protecting: deals with monitoring the environment for threats and responding to them. It is related to self-optimizing aspect but with the difference that the situation needs time-bound response and lead to cascading effect. Humans want to be closely involved based on the seriousness of the situation.

Table 1 summarizes the level to which information about the initial state (I), goals (G), action specification (A), existing plans (S) and domain constraints (Constraints) is expected to be available in the different AC scenarios. In self-configuring situation, actions may be scripts whose pre- and post-condition information may not be known and there may be no plans available *a priori*. In self-healing scenario, A is expected so that alternative plans could be explored. In self-optimizing and self-protecting scenarios, a plan would be available for the running system but the goal specification will be more clearly defined for the latter.

Type	I	G	A	S	Constraints
Self-configuring	Yes	Yes	-	-	Yes
Self-healing	Yes	Yes	Yes	-	Yes
Self-optimizing	-	-	-	Yes	Yes
Self-protecting	-	Yes	-	Yes	Yes

Table 1: The level to which planning problem information is expected to be available in AC scenarios. (-) means not assured.

From the above discussion, planning needs for AC can be summarized as follows:

1. The plan representation can be as general as workflows, e.g. BPEL4WS(Curbera et al 2002), with sequence, conditional, parallel, non-deterministic and loop constructs.
2. The plans are needed even if the initial state, goal state and action specification are not available, individually or collectively.
3. Automated plan generation is important but plans could also be obtained by users or domain-dependent methods. Even automatically generated plan may be analyzed by users before execution.
4. Over time, there would be a repository of previously generated and executed plans. They have to be considered while selecting existing or generating new plans.
5. The plans would typically be centrally executed but in large applications, the execution can be distributed.

CHAMPS(Keller et al 2004) is an example of a domain-dependent planner for AC self-configuration scenario while ABLE(Bigus et al 2002), extended with domain-independent Planner4J planners(Srivastava et al 2004), has been employed in self-healing scenarios. They are initial attempts to validate the potential of planning in AC.

AC Specific Challenges in using Planning

Based on our survey and experience of applying planning to autonomic computing, we identify two important challenges that AC applications pose to automated planning research – the need to support planning in partially specified domains, and the need to support plan life-cycle management. In this section, we describe these briefly.

Handling Incomplete Domain Model

The fact that a domain model is incomplete means many things. It could mean that domain is incompletely known though whatever is known is correct. This is orthogonal/different from expressiveness of domain model, e.g., PDDL levels(Fox & Long 2002), where the domain model at each level is complete though it may abstract some details of the world, which may get revealed in a more detailed higher PDDL level. Expressiveness impacts the complexity of planning and the representation of output plan. Incomplete domain model is also orthogonal/different from planning formulations varying in complexity like classical, conditional with partial observability, etc., where a problem is intrinsically of one type and hence cannot be expressed in any more simpler form.

If a plan is generated with incomplete domain models, it leads missing or under specified causal dependencies between actions in the plan. This affects the soundness guarantee of the planner because a generated plan may turn out to be not executable.

A domain may be incompletely specified in many ways. Formally, a planning domain is incomplete if at least one of the following happens (* denotes the corresponding complete specification):

- $P \subset P^*$
- $A_i^{pre} \subset A_i^{pre*}$ for some A_i
- $A_i^{post} \subset A_i^{post*}$ for some A_i
- There are relations $\alpha_i : \{P_i\} \times P$ which are not reflected in causal dependencies for achievement of predicates in A
- There are relations $\beta_i : \{A_i\} \times A$ which are not reflected in causal dependencies among actions in A

In the first case, the list of predicates in the domain is incomplete. In the second case, the list of preconditions for an action is incomplete. The preconditions can also be disjunctive but in contrast to traditional planning where disjunction is due to inherent uncertainty that will disappear at runtime, disjunction due to incomplete model will only get resolved with more domain input. In the third case, the list of postconditions of an action is incomplete and it can only get resolved with more domain input.

In many real domains like AC, the dependency among tasks or predicates is given but it is not explained in terms of a causal explanation (i.e., what precondition/effect dependencies are violated if the dependency is violated). For example, it is known that a specific action must occur before another action but this information is known as an ordering relation ($A_i \prec A_j$) but the actions do not have a causal dependency in terms of the modeled pre- and post conditions (Singh et al 1995). The fourth and fifth cases represent specification of dependencies among predicates and actions, respectively, that do not have causal explanation. Axioms can be used to specify these types of incompleteness.

The challenge for planning community is how to effectively deal with such incompleteness. There has been some initial work, e.g., in(Garland & Lesh 2002), the authors look at the problem of evaluating plans when the underlying actions are incompletely modeled. They define four types of risk based on the structure of the plan provided that any action's specification can be corrected in future. Plans are compared based on their assessed risks, and a ranking is derived. To plan with relations that do not have causal dependencies, techniques from the intersection of planning and distributed scheduling (c.f. (Beck & Fox 1999; Singh et al 1995)) will need to be adopted and extended.

Managing Life Cycle of Plans

A plan is synthesized for meeting some goals. But synthesis is just the beginning of a complex life-cycle management process. Plans must be organized in large collections, where they can be grouped along different purposes and are amenable to search, inspection, evaluation, and modification by human experts or automated reasoning systems. With users in the loop, plans which have been used in the past and have been successful, are more likely to be used again. New plans would get requested only when there is a deficiency in the existing plans. Eventually, plans will outlast their utility and be replaced.

Planning community has focused primarily on synthesis. To support AC applications, one needs to manage the life cycle of plans within an application and based on the context of their usage. For example, one needs techniques to automatically generate metadata annotations of plans that could be used for storage and retrieval. If humans provide metadata, each annotation could be different and metadata mismatches will become a critical issue unless the user is very constrained.

The challenges in generating metadata for managing plans are many. The plan can be as expressive as general workflows with both automated and manual sub-plans. The specification of the pre- and postconditions of each action may not be available. Furthermore, the initial situation for which the plan was generated and the goal it is supposed to achieve are seldomly available. This lack of information, which is taken for granted in AI planning, necessitates new techniques to deduce a plan's usage context. An initial approach for plan life cycle is discussed in (Srivastava et al 2005) where plan analysis techniques take BPEL4WS workflows or PDDL plans as input, build action models using plan structure and generate metadata based on the given plan and as well as compared to other plans in a plan repository.

Relationship with other technologies

We now discuss relationship of planning with procedural policies, and Web services and Grid.

Relationship between procedural policies and plans

As mentioned in the introduction, the term policy is used to refer to any declarative specification of behavior that is desired from a software system but they usually refer to procedural policies. There are many choices for a procedural policy language for AC, e.g., WS-Policy[1] being defined for web services and REI[2]. Most languages support variations of the Event-Condition-Action (ECA) specification. ECA rules specify what actions to take in response to events provided stated conditions hold, i.e., (Bailey et al 2002):

[1]ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf

[2]http://ebiquity.umbc.edu/v2.1/get/a/publication/57.pdf

On : $\prec$ *Event* $\succ$
If : $\prec$ *Condition* $\succ$ *holds*
Do : $\prec$ *Actions* $\succ$

Action refers to any activity that can be performed in the domain and a policy may consist of one or more actions. The policy language may additionally allow specification of scoping and priority (business value) of rules that can be used for rule selection while working with a set of rules.

Planning can be used for managing procedural policies - while creating new policies, validating properties with existing policies, updating policies - based on whether a set of policies could be composed. In (Srivastava 2004), it was shown how decision support problems in managing software components over their life cycle could be answered by posing them as planning problems. The same could be done for procedural policies.

Reciprocally, procedural policies can be used while planning for the AC scenarios. Essentially, they allow users to decide what decision to make in a situation, and this information can be used to pick any information needed for planning. More specifically, procedural policies can be used to:

- Select information for goals (G)
- Select information for initial state (I)
- Select actions relevant for planning (A) and what gets modeled in their specification.
- Select predicates in the planning problem (P).

Relating AC with Web Services and Scientific Flows

Planning is actively being applied for composition of web services(Srivastava & Koehler 2003) and scientific workflows (grid)(Blythe et al 2003). There are interesting similarities and contrasts between the planning requirements of autonomic computing and those of web services and scientific workflows. All of them require an expressive plan representation like BPEL4WS. All of these applications also pose the challenge of incomplete domain theories. In the case of web services, the incompleteness may come because of faulty or incomplete service annotations, while for workflows, the incompleteness may come because of constraints and dependencies without causal explanations. Plan management is also critical for these applications, while the need for automated synthesis is less prominent. In grid and web services, the plans will be distributedly executed while they will be primarily centrally executed in AC. Hence, techniques from distributed planning for generating concurrent plans are more relevant to the former.

Conclusion

In this paper, we explored the planning needs of AC, its match with existing planning technology, and its connections with policies and planning for web services and scientific workflows (grids). We observe that AC requirements call for plan synthesis and management techniques that work with incomplete domain specifications (theories) and support a life cycle view of plans.

Acknowledgements

We thank Jana Koehler, Joe Bigus, Joe Hellerstein and Jeff Kephart for useful discussions on practical planning.

References

Bailey, J., Poulovassilis, A., and Wood, P. 2002. An Event-Condition-Action Language for XML. *Proc. WWW, Honolulu, Hawaii.*

Beck, C., Fox, M. Scheduling Alternative Activities. AAAI/IAAI 1999: 680-687.

Bigus, J., Schlosnagle, D., Pilgrim, J., Mills, W., and Diao, Y. 2002. ABLE: A Toolkit for Building Multiagent Autonomic Systems. *IBM Systems Journal, Volume 41, Number 3. Also at http://www.research.ibm.com/journal/sj/413/bigus.html.*

Blythe, J., Deelman, E., Gil, Y., Kesselman, C., Agarwal, A., and Mehta, G. 2003. The Role of Planning in Grid Computing. *Proc. Intl Conf. on Automated Planning and Scheduling (ICAPS).*

Blythe, J. 1999. An Overview of Planning Under Uncertainty. *AI Magazine, Vol. 20(2), pp. 35–54.*

Curbera, F., and others. 2002. Business Process Execution Language for Web Services (BPEL4WS). *http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.*

Fox, M., and Long, D. 2002. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Available at http://www.dur.ac.uk/d.p.long/competition.html.*

Garland, A. and Lesh, N. 2002. Plan evaluation with incomplete action descriptions. *In Proc. of the 18th National Conference on Artificial intelligence (AAAI'02), USA.*

Kambhampati, S., Knoblock, C. and Yang, Q. 1995. Planning as Refinement Search: A Unifying framework for evaluating design trafeoffs in partial order planning. *Artificial Intelligence, Special issue on Planning and Scheduling. Vol. 76.*

Kambhampati, S., and Srivastava, B. 1995. Universal Classical Planner: An algorithm for unifying state space and plan space approaches. *In New Trend in AI Planning: EWSP 95, IOS Press.*

Keller, A., Hellerstein, J., Wolf, J., Wu, K. and Krishnan, V. 2004. The CHAMPS System: Change Management with Planning and Scheduling. *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS 2004).*

Kephart, J. and Chess, D. 2003. The Vision of Autonomic Computing. *IEEE Computer, Vol. 36, No. 1, pp 41-50.*

Munindar P. Singh, Greg Meredith, Christine Tomlinson, Paul C. Attie: An Event Algebra for Specifying and Scheduling Workflows. DASFAA 1995: 53-60.

Srivastava, B. 2004. A Decision-support Framework for Component Reuse and Maintenance in Software Project Management. *IEEE 8th European Conference on Software Maintenance and Reengineering (CSMR 2004), Tampere, Finland.*

Srivastava, B., Bigus, J., and Schlosnagle, D. 2004. Bringing Planning to Autonomic Applications with ABLE. *Proc. IEEE 1st International Conference on Autonomic Computing, New York, USA.*

Srivastava, B. and Koehler, J. 2003. Web Service Composition: Current Solutions and Open Problems. *ICAPS 2003 Workshop on Planning for Web Services, pages 28 - 35.*

Srivastava, B., Vanhatalo, J., and Koehler, J. 2005. Managing the Life Cycle of Plans. *Proc. IAAI/AAAI.*

Weld, D. 1999. Recent Advances in AI Planning. *AI Magazine,* Volume 20, No.2, pp 93-123.