



# **14<sup>th</sup> International Conference on Automated Planning and Scheduling**

Whistler, British Columbia, Canada - June 3-7, 2004

## **International Planning Competition**

### **Chairs Classical Track**

Stefan Edelkamp, University of Dortmund (Germany)

Jörg Hoffmann, Albert-Ludwigs-University (Germany)

### **Chairs Probabilistic Track**

Michael Littman, Rutgers University (USA)

Håkan Younes, Carnegie Mellon University (USA)



14<sup>th</sup> International Conference on Automated Planning and Scheduling  
June 3-7, 2004 in Whistler, British Columbia, Canada  
Conference Chairs: Shlomo Zilberstein, Jana Koehler, Sven Koenig

## **International Planning Competition**

Chairs Classical Track

Stefan Edelkamp, University of Dortmund (Germany)

Jörg Hoffmann, Albert-Ludwigs-University (Germany)

Chairs Probabilistic Track

Michael Littman, Rutgers University (USA)

Håkan Younes, Carnegie Mellon University (USA)

The material in these notes is copyrighted by its respective authors.  
It does not count as published.

For more information on ICAPS, please visit [www.icaps-conference.org](http://www.icaps-conference.org).



## Preface

From a research perspective, running a competition pushes the envelope in the development and implementation of new or improved algorithms and data structures. The fourth international planning competition, IPC-4 for short, has attracted many competitors, and we as the organisers hope that the event will be a significant step in promoting the acceptance and applicability of planning technology.

The competition and its organisation is split into two parts. On the one hand, there is the *classical* part that, in continuation of the previous competition events, considers “classical” fully deterministic and observable planning. On the other hand, there is – for the first time in the history of the event – a *probabilistic* part, featuring factored encodings of fully observable Markov decision problems. In both parts, variations of PDDL as the common language lay the basis for the competition.

The 4th IPC has several exciting aspects. On the one hand, the classical track features more realistic benchmark domains, formulated (in part) with the help of two new language extensions. There is an extra track for optimal planners (planners that give a guarantee on the quality of the returned solution), and with round about 20 competing systems the event is even a little larger than its already large predecessors. The existence of the probabilistic part is, of course, exciting in itself. It is a great success in that it also attracted several competing systems, since the probabilistic competition is completely new!

Talking about competing systems, the organisers wish to say a big “thank you” to all the participating teams for their efforts. There is significant bravery in the submission of a planning system to a competition, where the choice and design of the benchmark problems is up to the competition organisers, not to the individuals!

It is the first time that a booklet like this is distributed at the host conference. The organisers hope that, with this booklet, the transparency and understandability of the competition event, at the time of its happening (or at least shortly after), will greatly improve, given that over 60 authors have contributed to it. The actual results of the competition are, of course, not yet collected at the time of writing. The results will be made available at ICAPS’04 in the form of posters that will be put up in the coffee break room.

The booklet is divided into two parts, one about the classical part of IPC-4, one about the probabilistic part. Both parts contain extended abstracts written by participating teams, describing their planner or their planners – each team was allowed to enter (at most) two competing systems. Note that the abstracts were written while the competition was still running, so the abstracts might not describe the full functionalities of the final system versions. Each part of the booklet also includes a brief presentation of the PDDL variant used. For the classical part we have added an extra abstract giving short description of our benchmark domains, to give people an idea of what kinds of problems the

planners were tested on, and how we created these problems.

We hope that, by reading this booklet, everybody receives an impression of the the fun, importance and charme of this year's competition event. We wish all of you an exciting conference!

Stefan Edelkamp and Jörg Hoffmann (co-chairs classical track)

Michael Littman and Håkan L. S. Younes (co-chairs probabilistic track)



# Table of Contents

## Classical Part

PDDL2.2: The Language for the Classical Part of IPC-4 <i>Stefan Edelkamp and Jörg Hoffmann</i> .....	2
Towards Realistic Benchmarks for Planning: the Domains used in the Classical Part of IPC-4 <i>Jörg Hoffmann, Stefan Edelkamp, Roman Englert, Frederico Liporace, Sylvie Thiébaux, and Sebastian Trüg</i> .....	7
Macro-FF <i>Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer</i> ..	15
Optiplan: Unifying IP-based and Graph-based Planning <i>Menkes van den Briel and Subbarao Kambhampati</i> .....	18
FAP: Forward Anticipating Planner <i>Guy Camilleri and Joseph Zalaket</i> .....	21
Marvin: Macro Actions from Reduced Versions of the Instance <i>Andrew Coles and Amanda Smith</i> .....	24
A Petri net based representation for planning problems <i>Marcos Casilho and André Guedes, Tiago Lima, João Marynowski, Razer Montaña, Luis Künzle, and Fabiano Silva</i> .....	27
SGPlan: Subgoal Partitioning and Resolution in Planning <i>Yixin Chen, Chih-Wei Hsu, and Benjamin W. Wah</i> .....	30
Planning in PDDL2.2 Domains with LPG-TD <i>Alfonso Gerevini, Alessandro Saetti, Ivan Serina, and Paolo Toninelli</i> .....	33
The Working of CRIKEY – a Temporal Metric Planner <i>Keith Halsey</i> .....	35
TP4'04 and HSP* <sub>a</sub> <i>Patrik Haslum</i> .....	38
Fast Downward – Making use of causal dependencies in the problem representation <i>Malte Helmert and Silvia Richter</i> .....	41



SATPLAN04: Planning as Satisfiability <i>Henry Kautz</i> .....	44
Tilsapa - Timed Initial Literals Using SAPA <i>Bharat Ranjan Kavuluri and Senthil U</i> .....	46
The Optop Planner <i>Drew McDermott</i> .....	48
Combining Backward-Chaining With Forward-Chaining AI Search <i>Eric Parker</i> .....	51
P-MEP: Parallel More Expressive Planner <i>Javier Sanchez, Minh Tang and Amol D. Mali</i> .....	53
The YAHSP planning system: Forward heuristic search with lookahead plans analysis <i>Vincent Vidal</i> .....	56
CPT: An Optimal Temporal POCL Planner based on Constraint Programming <i>Vincent Vidal and Héctor Geffner</i> .....	59
BFHSP: A Breadth-First Heuristic Search Planner <i>Rong Zhou and Eric A. Hansen</i> .....	61
Heuristic Planning via Roadmap Deduction <i>Lin Zhu and Robert Givan</i> .....	64

## Probabilistic Part

Introduction to the Probabilistic Track <i>Michael Littman and Håkan L. S. Younes</i> .....	68
PPDDL1.0: The Language for the Probabilistic Part of IPC-4 <i>Håkan L. S. Younes and Michael Littman</i> .....	70
mGPT: A Probabilistic Planner based on Heuristic Search <i>Blai Bonet and Héctor Geffner</i> .....	74
Symbolic Heuristic Search for Probabilistic Planning <i>Zhengzhu Feng and Eric A. Hansen</i> .....	77
NMRDPP: Decision-Theoretic Planning with Control Knowledge <i>Charles Gretton, David Price, and Sylvie Thiébaux</i> .....	80
FCPlanner: A Planning Strategy for First-Order MDPs <i>Eldar Karabaev, and Olga Skvortsova</i> .....	83
Probapop: Probabilistic Partial-Order Planning <i>Nilufer Onder, Garrett C. Whelan, and Li Li</i> .....	86
Probabilistic Reachability Analysis for Structured Markov Decision Processes <i>Florent Teichteil-Königsbuch and Patrick Fabiani</i> .....	89
Learning Reactive Policies for Probabilistic Planning Domains <i>SungWook Yoon, Alan Fern, and Robert Givan</i> .....	92

## Classical Part

# PDDL2.2: The Language for the Classical Part of IPC-4

## — extended abstract —

**Stefan Edelkamp**

Fachbereich Informatik  
Baroper Str. 301, GB IV  
44221 Dortmund, Germany  
stefan.edelkamp@cs.uni-dortmund.de

**Jörg Hoffmann**

Institut für Informatik  
Georges-Köhler-Allee, Geb. 52  
79110 Freiburg, Germany  
hoffmann@informatik.uni-freiburg.de

### Introduction

The 3rd International Planning Competition, IPC-3, was run by Derek Long and Maria Fox. The competition focussed on planning in temporal and metric domains. For that purpose, Fox and Long developed the PDDL2.1 language (Fox & Long 2003), of which the first three *levels* were used in IPC-3. Level 1 was the usual STRIPS and ADL planning, level 2 added numeric variables, level 3 added durational constructs.

In this document, we describe the language, named *PDDL2.2*, used for formulating the domains used in the classical part of IPC-4. As the language extensions made for IPC-3 still provide major challenges to the planning community, the language extensions for IPC-4 are relatively moderate. The first three levels of PDDL2.1 are interpreted as an agreed fundament, and kept as the basis of PDDL2.2. PDDL2.2 also inherits the separation into the three levels. The language features added on top of PDDL2.1 are *derived predicates* (into levels 1, 2, and 3) and *timed initial literals* (into level 3 only). Both of these constructs are practically motivated, and are put to use in some of the competition domains. Details on the constructs are in the respective sections.

The next section discusses derived predicates, including a brief description of their syntax, and the definition of their semantics. The section after that does the same for timed initial literals. Full details, including a BNF description of PDDL2.2, can be found in a technical report (Edelkamp & Hoffmann 2004).

### Derived Predicates

Derived predicates have been implemented in several planning systems in the past, including e.g. UCPOP (Penberthy & Weld 1992). They are predicates that are not affected by any of the actions available to the planner. Instead, the predicate's truth values are derived by a set of rules of the form **if**  $\phi(\bar{x})$  **then**  $P(\bar{x})$ . The semantics are, roughly, that an instance of a derived predicate (a derived predicate whose arguments are instantiated with constants; a *fact*, for short) is TRUE iff it can be derived using the available rules (more details below). Under the name “axioms”, derived predicates were a part of the original PDDL language defined by McDermott (McDermott & others 1998) for the first planning competition, but they have never been put to use in a

competition benchmark (we use the name “derived predicates” instead of “axioms” in order to avoid confusion with safety conditions).

### Syntax

The BNF definition of derived predicates involves just two small modifications to the BNF definition of PDDL2.1:

```
<structure-def> ::= :derived-predicates  
                  <derived-def>
```

The domain file specifies a list of “structures”. In PDDL2.1 these were either actions or durational actions. Now we also allow “derived” definitions at these points.

```
<derived-def> ::= (:derived <atomic  
                  formula(term)> <GD>)
```

The “derived” definitions are the “rules” mentioned above. They simply specify the predicate  $P$  to be derived (with variable vector  $\bar{x}$ ), and the formula  $\phi(\bar{x})$  from which instances of  $P$  can be concluded to be true. Syntactically, the predicate and variables are given by the `<atomic formula(term)>` expression, and the formula is given by `<GD>` (a “goal description”, i.e. a formula).

The BNF is more generous than what we actually allow in PDDL2.2, respectively in IPC-4. We make a number of restrictions to ensure that the definitions make sense and are easy to treat algorithmically. We call a predicate  $P$  *derived* if there is a rule that has a predicate  $P$  in its head; otherwise we call  $P$  *basic*. The restrictions we make are the following.

1. The actions available to the planner do not affect the derived predicates: no derived predicate occurs on any of the effect lists of the domain actions.
2. If a rule defines that  $P(\bar{x})$  can be derived from  $\phi(\bar{x})$ , then the variables in  $\bar{x}$  are pairwise different (and, as the notation suggests, the free variables of  $\phi(\bar{x})$  are exactly the variables in  $\bar{x}$ ).
3. If a rule defines that  $P(\bar{x})$  can be derived from  $\phi$ , then the Negation Normal Form (NNF) of  $\phi(\bar{x})$  does not contain any derived predicates in negated form.

The first restriction ensures that there is a separation between the predicates that the planner can affect (the basic predicates) and those (the derived predicates) whose truth

values follow from the basic predicates. The second restriction ensures that the rule right hand sides match the rule left hand sides. Let us explain the third restriction. The NNF of a formula is obtained by “pushing the negations downwards”, i.e. transforming  $\neg\forall x : \phi$  into  $\exists x : (\neg\phi)$ ,  $\neg\exists x : \phi$  into  $\forall x : (\neg\phi)$ ,  $\neg\bigvee \phi_i$  into  $\bigwedge (\neg\phi_i)$ , and  $\neg\bigwedge \phi_i$  into  $\bigvee (\neg\phi_i)$ . Iterating these transformation steps, one ends up with a formula where negations occur only in front of atomic formulas – predicates with variable vectors, in our case. The formula contains a predicate  $P$  in *negated form* iff there is an occurrence of  $P$  that is negated. By requiring that the formulas in the rules (that derive predicate values) do not contain any derived predicates in negated form, we ensure that there can not be any negative interactions between applications of the rules (see the semantics below).

An example of a derived predicate is the “above” predicate in the *Blocksworld*, which is true between blocks  $x$  and  $y$  whenever  $x$  is transitively (possibly with some blocks in between) on  $y$ . Using the derived predicates syntax, this predicate can be defined as follows.

```
(:derived (above ?x ?y)
  (or (on ?x ?y)
    (exists (?z) (and (on ?x ?z)
                      (above ?z ?y))))))
```

Note that formulating the truth value of “above” in terms of the effects of the normal *Blocksworld* actions is very awkward (the unconvinced reader is invited to try). The predicate is the transitive closure of the “on” relation.

## Semantics

We now describe the updates that need to be made to the PDDL2.1 semantics definitions given by Fox and Long in (Fox & Long 2003). We introduce formal notations to capture the semantics of derived predicates. We then “hook” these semantics into the PDDL2.1 language by modifying two of the definitions in (Fox & Long 2003).

Say we are given the truth values of all (instances of the) basic predicates, and want to compute the truth values of the (instances of the) derived predicates from that. We are in this situation every time we have applied an action, or parallel action set. (In the durational context, we are in this situation at the “happenings” in our current plan, that is every time a durative action starts or finishes.) Formally, what we want to have is a function  $\mathcal{D}$  that maps a set of basic facts (instances of basic predicates) to the same set but enriched with derived facts (the derivable instances of the derived predicates). Assume we are given the set  $R$  of rules for the derived predicates, where the elements of  $R$  have the form  $(P(\bar{x}), \phi(\bar{x}))$  – **if**  $\phi(\bar{x})$  **then**  $P(\bar{x})$ . Then  $\mathcal{D}(s)$ , for a set of basic facts  $s$ , is defined as follows.

$$\mathcal{D}(s) := \bigcap \{s' \mid s \subseteq s', \forall (P(\bar{x}), \phi(\bar{x})) \in R : \forall \bar{c}, |\bar{c}| = |\bar{x}| : (s' \models \phi(\bar{c}) \Rightarrow P(\bar{c}) \in s')\}$$

This definition uses the standard notations of the modelling relation  $\models$  between states (represented as sets of facts in our case) and formulas, and of the substitution  $\phi(\bar{c})$  of the free variables in formula  $\phi(\bar{x})$  with a constant vector  $\bar{c}$ . In words,  $\mathcal{D}(s)$  is the intersection of all supersets of  $s$  that are closed under application of the rules  $R$ .

Remember that we restrict the rules to not contain any derived predicates in negated form. This implies that the order in which the rules are applied to a state does not matter (we can not “lose” any derived facts by deriving other facts first). This, in turn, implies that  $\mathcal{D}(s)$  is itself closed under application of the rules  $R$ . In other words,  $\mathcal{D}(s)$  is the least fixed point over the possible applications of the rules  $R$  to the state where all derived facts are assumed to be FALSE (represented by their not being contained in  $s$ ).

More constructively,  $\mathcal{D}(s)$  can be computed by the following simple process.

$s' := s$

**do**

**select** a rule  $(P(\bar{x}), \phi(\bar{x}))$  and a vector  $\bar{c}$  of constants,  $|\bar{c}| = |\bar{x}|$ , such that  $s' \models \phi(\bar{c})$

let  $s' := s' \cup \{P(\bar{c})\}$

**until** no rule and constant vector could be selected

let  $\mathcal{D}(s) := s'$

In words, apply the applicable rules in an arbitrary order until no new facts can be derived anymore.

We can now specify what an executable plan is in PDDL2.1 with derived predicates. All we need to do is to hook the function  $\mathcal{D}$  into Definition 13, “Happening Execution”, in (Fox & Long 2003). By this definition, Fox and Long define the state transitions in a plan. The happenings in a (temporal or non-temporal) plan are all time points at which at least one action effect occurs. Fox and Long’s definition is this:

**Definition 13 Happening Execution** (Fox and Long (2003))

Given a state,  $(t, s, \mathbf{x})$  and a happening,  $H$ , the activity for  $H$  is the set of grounded actions

$$A_H = \{a \mid \text{the name for } a \text{ is in } H, a \text{ is valid and } Pre_a \text{ is satisfied in } (t, s, \mathbf{x})\}$$

The result of executing a happening,  $H$ , associated with time  $t_H$ , in a state  $(t, s, \mathbf{x})$  is undefined if  $|A_H| \neq |H|$  or if any pair of actions in  $A_H$  is mutex. Otherwise, it is the state  $(t_H, s', \mathbf{x}')$  where

$$s' = (s \setminus \bigcup_{a \in A_H} Del_a) \cup \bigcup_{a \in A_H} Add_a \quad (**)$$

and  $\mathbf{x}'$  is the result of applying the composition of the functions  $\{NPF_a \mid a \in A_H\}$  to  $\mathbf{x}$ .

Note that the happenings consist of grounded actions, i.e. all operator parameters are instantiated with constants. To introduce the semantics of derived predicates, we now modify the result of executing the happening. (We will also adapt the definition of mutex actions, see below.) The result of executing the happening is now obtained by applying the actions to  $s$ , then subtracting all derived facts from this, then applying the function  $\mathcal{D}$ . That is, in the above definition we replace  $(**)$  with the following:

$$s' = \mathcal{D}((s \setminus \bigcup_{a \in A_H} Del_a) \cup \bigcup_{a \in A_H} Add_a \setminus D)$$

where  $D$  denotes the set of all derived facts. If there are no derived predicates,  $D$  is the empty set and  $\mathcal{D}$  is the identity function.

As an example, say we have a *Blocksworld* instance where  $A$  is on  $B$  is on  $C$ ,  $s = \{clear(A), on(A, B), on(B, C), ontable(C), above(A, B), above(B, C), above(A, C)\}$ , and our happening applies an action that moves  $A$  to the table. Then the happening execution result will be computed by removing  $on(A, B)$  from  $s$ , adding  $clear(B)$  and  $ontable(A)$  into  $s$ , removing all of  $above(A, B)$ ,  $above(B, C)$ , and  $above(A, C)$  from  $s$ , and applying  $\mathcal{D}$  to this, which will re-introduce (only)  $above(B, C)$ . So  $s'$  will be  $s' = \{clear(A), ontable(A), clear(B), on(B, C), ontable(C), above(B, C)\}$ .

By the definition of happening execution, Fox and Long (Fox & Long 2003) define the state transitions in a plan. The definitions of what an executable plan is, and when a plan achieves the goal, are then standard. The plan is *executable* if the result of all happenings in the plan is defined. This means that all action preconditions have to be fulfilled in the state of execution, and that no two pairs of actions in a happening are *mutex*. The plan *achieves the goal* if the goal holds true in the state that results after the execution of all actions in the plan.

With our above extension of the definition of happening executions, the definitions of plan executability and goal achievement need not be changed. We do, however, need to adapt the definition of when a pair of actions is *mutex*. This is important if the happenings can contain more than one action, i.e. if we consider parallel (e.g. Graphplan-style) or concurrent (durational) planning. Fox and Long (Fox & Long 2003) give a conservative definition that forbids the actions to interact in any possible way. The definition is the following.

**Definition 12 Mutex Actions** (Fox and Long (2003))

Two grounded actions,  $a$  and  $b$  are non-interfering if

$$\begin{aligned} GPre_a \cap (Add_b \cup Del_b) &= GPre_b \cap (Add_a \cup Del_a) = \emptyset (*) \\ Add_a \cap Del_b &= Add_b \cap Del_a = \emptyset \\ L_a \cap R_b &= R_a \cap L_b = \emptyset \\ L_a \cap L_b &\subseteq L_a^* \cup L_b^* \end{aligned}$$

If two actions are not non-interfering they are *mutex*.

Note that the definition talks about grounded actions where all operator parameters are instantiated with constants.  $L_a$ ,  $L_b$ ,  $R_a$ , and  $R_b$  refer to the left and right hand side of  $a$ 's and  $b$ 's numeric effects.  $Add_a/Add_b$  and  $Del_a/Del_b$  are  $a$ 's and  $b$ 's positive (add) respectively negative (delete) effects.  $GPre_a/Gpre_b$  denotes all (ground) facts that occur in  $a$ 's/ $b$ 's precondition. If a precondition contains quantifiers then these are grounded out ( $\forall x$  transforms to  $\bigwedge c_i$ ,  $\exists x$  transforms to  $\bigvee c_i$  where the  $c_i$  are all objects in the given instance), and  $GPre$  is defined over the resulting quantifier-free (and thus variable-free) formula. Note that this definition of mutex actions is very conservative – if, e.g., fact  $F$  occurs only positively in  $a$ 's precondition, then it does not matter if  $F$  is among the add effects of  $b$ . The conservative definition has the advantage that it makes it algorithmically very easy to figure out if or if not  $a$  and  $b$  are *mutex*.

In the presence of derived predicates, the above definition needs to be extended to exclude possible interactions that can arise indirectly due to derived facts, in the precondition of the one action, whose truth value depends on the truth value of (basic) facts affected by the effects of the other action. In the same spirit in that Fox and Long forbid any possibility of direct interaction, we now forbid any possibility of indirect interaction. Assume we ground out all rules  $(P(\bar{x}), \phi(\bar{x}))$  for the derived predicates, i.e. we insert all possible vectors  $\bar{c}$  of constants; we also ground out the quantifiers in the formulas  $\phi(\bar{c})$ , ending up with variable free rules. We define a directed graph where the nodes are (ground) facts, and an edge from fact  $F$  to fact  $F'$  is inserted iff there is a grounded rule  $(P(\bar{c}), \phi(\bar{c}))$  such that  $F' = P(\bar{c})$ , and  $F$  occurs in  $\phi(\bar{c})$ . Now say we have an action  $a$ , where all ground facts occurring in  $a$ 's precondition are, see above, denoted by  $GPre_a$ . By  $DPre_a$  we denote all ground facts that can possibly influence the truth values of the derived facts in  $GPre_a$ :

$$DPre_a := \{F \mid \text{there is a path from } F \text{ to an } F' \in GPre_a\}$$

The definition of *mutex* actions is now updated simply by replacing, in the above definition,  $(***)$  with:

$$\begin{aligned} (DPre_a \cup GPre_a) \cap (Add_b \cup Del_b) &= \\ (DPre_b \cup GPre_b) \cap (Add_a \cup Del_a) &= \emptyset \end{aligned}$$

As an example, reconsider the *Blocksworld* and the “above” predicate. Assume that the action that moves a block  $A$  to the table requires as an additional, derived, precondition, that  $A$  is above some third block. Then, in principle, two actions that move two different blocks  $A$  and  $B$  to the table can be executed in parallel. Which block  $A$  ( $B$ ) is on can influence the *above* relations in that  $B$  ( $A$ ) participates; however, this does not matter because if  $A$  and  $B$  can be both moved then this implies that they are both clear, which implies that they are on top of different stacks anyway. We observe that the latter is a statement about the domain semantics that either requires non-trivial reasoning, or access to the world state in which the actions are executed. In order to avoid the need to either do non-trivial reasoning about domain semantics, or resort to a forward search, our definition is the conservative one given above. The definition makes the actions moving  $A$  and  $B$  *mutex* on the grounds that they can possibly influence each other's derived preconditions.

The definition adaptations described above suffice to define the semantics of derived predicates for the whole of PDDL2.2. Fox and Long reduce the temporal case to the case of simple plans above, so by adapting the simple-plan definitions we have automatically adapted the definitions of the more complex cases. In the temporal setting, PDDL2.2 level 3, the derived predicates semantics are that their values are computed anew at each happening in the plan where an action effect occurs.

## Timed Initial Literals

Timed initial literals are a syntactically very simple way of expressing a certain restricted form of exogenous events: facts that will become TRUE or FALSE at time points that are known to the planner in advance, independently of the

actions that the planner chooses to execute. Timed initial literals are thus deterministic unconditional exogenous events. Syntactically, we simply allow the initial state to specify – beside the usual facts that are true at time point 0 – literals that will become true at time points greater than 0.

Timed initial literals are practically very relevant: in the real world, deterministic unconditional exogenous events are very common, typically in the form of time windows (within which a shop has opened, within which humans work, within which traffic is slow, within which there is daylight, within which a seminar room is occupied, within which nobody answers their mail because they are all at conferences, etc.).

## Syntax

As said, the syntax simply allows literals with time points in the initial state.

```
<init> ::= (:init <init-el>*)
<init-el> ::= :timed-initial-literals (at <number>
                                     <literal(name)>)
```

The requirement flag for timed initial literals implies the requirement flag for durational actions, i.e. as said the language construct is only available in PDDL2.2 level 3. The times <number> at which the timed literals occur are restricted to be greater than 0. If there are also derived predicates in the domain, then the timed literals are restricted to not influence any of these, i.e., like action effects they are only allowed to affect the truth values of the basic (non-derived) predicates (IPC-4 will not use both derived predicates and timed initial literals within the same domain).

As an illustrative example, consider a planning task where the goal is to be done with the shopping. There is a single action *go-shopping* that achieves the goal, and requires the (single) shop to be open as the precondition. The shop opens at time 9 relative to the initial state, and closes at time 20. We can express the shop opening times by two timed initial literals:

```
(:init
 (at 9 (shop-open))
 (at 20 (not (shop-open)))
)
```

## Semantics

We now describe the updates that need to be made to the PDDL2.1 semantics definitions given by Fox and Long in (Fox & Long 2003). Adapting two of the definitions suffices.

The first definition we need to adapt is the one that defines what a “simple plan”, and its happening sequence, is. The original definition by Fox and Long is this.

### Definition 11 Simple Plan (Fox and Long (2003))

A simple plan,  $SP$ , for a planning instance,  $I$ , consists of a finite collection of timed simple actions which are pairs  $(t, a)$ , where  $t$  is a rational-valued time and  $a$  is an action name.

The happening sequence,  $\{t_i\}_{i=0\dots k}$  for  $SP$  is the ordered sequence of times in the set of times appearing in the timed

simple actions in  $SP$ . All  $t_i$  must be greater than 0. It is possible for the sequence to be empty (an empty plan).

The happening at time  $t$ ,  $E_t$ , where  $t$  is in the happening sequence of  $SP$ , is the set of (simple) action names that appear in timed simple actions associated with the time  $t$  in  $SP$ .

In the STRIPS case, the time stamps are the natural numbers  $1, \dots, n$  when there are  $n$  actions/parallel action sets in the plan. The happenings then are the actions/parallel action sets at the respective time steps. Fox and Long reduce the temporal planning case to the simple plan case defined here by splitting each durational action up into at least two simple actions – the start action, the end action, and possibly several actions in between that guard the durational action’s invariants at the points where other action effects occur. So in the temporal case, the happening sequence is comprised of all time points at which “something happens”, i.e. at which some action effect occurs.

To introduce our intended semantics of timed initial literals, all we need to do to this definition is to introduce additional happenings into the temporal plan, namely the time points at which some timed initial literal occurs. The timed initial literals can be interpreted as simple actions that are forced into the respective happenings (rather than selected into them by the planner), whose precondition is true, and whose only effect is the respective literal. The rest of Fox and Long’s definitions then carry over directly (except goal achievement, which involves a little care, see below). The PDDL2.2 definition of simple plans is this here.

### Definition 11 Simple Plan

A simple plan,  $SP$ , for a planning instance,  $I$ , consists of a finite collection of timed simple actions which are pairs  $(t, a)$ , where  $t$  is a rational-valued time and  $a$  is an action name. By  $t_{end}$  we denote the largest time  $t$  in  $SP$ , or 0 if  $SP$  is empty.

Let  $TL$  be the (finite) set of all timed initial literals, given as pairs  $(t, l)$  where  $t$  is the rational-valued time of occurrence of the literal  $l$ . We identify each timed initial literal  $(t, l)$  in  $TL$  with a uniquely named simple action that is associated with time  $t$ , whose precondition is *TRUE*, and whose only effect is  $l$ .

The happening sequence,  $\{t_i\}_{i=0\dots k}$  for  $SP$  is the ordered sequence of times in the set of times appearing in the timed simple actions in  $SP$  and  $TL$ . All  $t_i$  must be greater than 0. It is possible for the sequence to be empty (an empty plan).

The happening at time  $t$ ,  $E_t$ , where  $t$  is in the happening sequence of  $SP$ , is the set of (simple) action names that appear in timed simple actions associated with the time  $t$  in  $SP$  or  $TL$ .

Thus the happenings in a temporal plan are all points in time where either an action effect, or a timed literal, occurs. The timed literals are simple actions forced into the plan. With this construction, Fox and Long’s Definitions 12 (Mutex Actions) and 13 (Happening Execution), as described (and adapted to derived predicates) in Section , can be kept unchanged. They state that no action effect is allowed to interfere with a timed initial literal, and that the timed initial

literals are true in the state that results from the execution of the happening they are contained in. Fox and Long’s Definition 14 (Executability of a plan) can also be kept unchanged – the timed initial literals change the happenings in the plan, but not the conditions under which a happening can be executed.

The only definition we need to re-think is that of what the *makespan* of a valid plan is. In Fox and Long’s original definition, this is implicit in the definition of valid plans. The definition is this.

**Definition 15 Validity of a Simple Plan** (Fox and Long (2003))

*A simple plan (for a planning instance,  $I$ ) is valid if it is executable and produces a final state  $S$ , such that the goal specification for  $I$  is satisfied in  $S$ .*

The makespan of the valid plan is accessible in PDDL2.1 and PDDL2.2 by the “total-time” variable that can be used in the optimization expression. Naturally, Fox and Long take the makespan to be the end of the plan, the time point of the plan’s final state.

In the presence of timed initial literals, the question of what the plan’s makespan is becomes a little more subtle. With Fox and Long’s above original definition, the makespan would be the end of all happenings in the simple plan, which *include* all timed initial literals (see the revised Definition 11 above). So the plan would at least take as long as it takes until no more timed literals occur. But a plan might be finished long before that – imagine something that needs to be done while there is daylight; certainly the plan does not need to wait until sunset. We therefore define the makespan to be the earliest point in time at which the goal condition becomes (and remains) true. Formally this reads as follows.

**Definition 15 Validity and Makespan of a Simple Plan**

*A simple plan (for a planning instance,  $I$ ) is valid if it is executable and produces a final state  $S$ , such that the goal specification for  $I$  is satisfied in  $S$ . The plan’s makespan is the smallest  $t \geq t_{end}$  such that, for all happenings at times  $t' \geq t$  in the plan’s happening sequence, the goal specification is satisfied after execution of the happening.*

Remember that  $t_{end}$  denotes the time of the last happening in the plan that contains an effect caused by the plan’s actions – in simpler terms,  $t_{end}$  is the end point of the plan. What the definition says is that the plan is valid if, at some time point  $t$  after the plan’s end, the goal condition is achieved and remains true until after the last timed literal has occurred. The plan’s makespan is the first such time point  $t$ . Note that the planner can “use” the events to achieve the goal, by doing nothing until a timed literal occurs that makes the goal condition true – but then the waiting time until the nearest such timed literal is counted into the plan’s makespan. (The latter is done to avoid situations where the planner could prefer to wait millions of years rather than just applying a single action itself.) Remember that the makespan of the plan, defined as above, is what can

be denoted by `total-time` in the optimization expression defined with the problem instance.

**Acknowledgements.** We would like to thank the IPC-4 organizing committee for their help in taking the decision about the language for the classical part of IPC-4, and in ironing out the details about syntax and semantics. The people contributing to this discussion were Drew McDermott, Daniel Weld, David Smith, Hakan Younes, Jussi Rintanen, Sylvie Thiebaux, Maria Fox, and Derek Long. We especially thank Maria Fox and Derek Long for giving us the latex sources of their PDDL2.1 article, and for discussing the modifications of this document needed to introduce the semantics of derived predicates and timed initial literals.

## References

- Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Albert-Ludwigs-Universität, Institut für Informatik, Freiburg, Germany.
- Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*. Special issue on the 3rd International Planning Competition, to appear.
- McDermott, D., et al. 1998. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Committee.
- Penberthy, J. S., and Weld, D. S. 1992. UCPOP: A sound, complete, partial order planner for ADL. In Nebel, B.; Swartout, W.; and Rich, C., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the 3rd International Conference (KR-92)*, 103–114. Cambridge, MA: Morgan Kaufmann.
- Thiebaux, S.; Hoffmann, J.; and Nebel, B. 2003. In defense of PDDL axioms. In Gottlob, G., ed., *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*. Acapulco, Mexico: Morgan Kaufmann. accepted for publication.



# Towards Realistic Benchmarks for Planning: the Domains Used in the Classical Part of IPC-4 – extended abstract –

Jörg Hoffmann\*

Stefan Edelkamp†

Roman Englert‡

Frederico Liporace§

Sylvie Thiébaux¶

Sebastian Trüg||

## Introduction

Today, the research discipline of AI planning is largely concerned with improving the performance of general problem solving mechanisms. Performance is measured by testing systems on example instances of the problem to be solved. Clearly, since no mechanism will ever be able to perform well on *all* instances of a (hard) problem, one of the most crucial issues in such a research context is what kind of examples are used for the testing. Add on top of this that, more and more, researchers draw their testing examples from the collections used in the IPC, and it becomes evident that the IPC benchmarks are nowadays one of the most important instruments for the field.

In the organisation of the (classical part of the) 4th IPC, we therefore invested considerable effort into creating a set of “appropriate” benchmarks for planning. The criteria applied for appropriateness were that the benchmarks should be:

1. **Oriented at applications** – a benchmark should reflect an application that the field is heading for.
2. **Diverse in structure** – a set of benchmarks should cover different kinds of structure that can occur in the attacked problem.
3. **Suitable for basic research** – a set of benchmarks for a field of basic research should not omit the basic aspects of that research.

The first of these criteria is probably the one most widely agreed upon – indeed, AI planning has frequently been criticised for its “obsession with toy examples”. In recent years, the performance of state-of-the-art systems has improved dramatically, and with that more realistic examples came within reach. We made another step in this direction by

orienting most of the IPC-4 benchmarks at application domains. While traditionally planning benchmarks were more or less phantasy products created having some “real” scenario in mind, we took actual (possible) applications of planning technology, and turned them into something suitable for the competition. In the process of adapting an application for use in the (current form of the) IPC, inevitably some of the realism has to give way to more pragmatic considerations (expected planner performance, language capabilities, etc.). Nevertheless, we believe that the IPC-4 domains are a significant step into the right direction.

The second of the above listed appropriateness criteria has traditionally been given less attention than the first one, but we believe that it is not less important. The structure underlying a testing example determines the performance of the applied solving mechanism. This is particularly true for solving mechanisms whose performance rises and falls with the quality of a heuristic they use. Hoffmann (2002)’s results suggest that much of the spectacular performance of modern heuristic search planners is due to structural similarities between most of the traditional planning benchmarks. While this does by no means imply that modern heuristic search planners aren’t useful, it certainly shows that in the creation of benchmarks there is a risk of introducing a bias towards one specific way of solving them. In selecting the benchmark domains for IPC-4, we took care to cover a range of intuitively very different kinds of problem structure.<sup>1</sup>

Finally, the third of our appropriateness criteria is probably agreed on by nobody – except all the people whose planners can only handle STRIPS. More seriously, we believe that, with all the new PDDL extensions, the planning community ought to not let completely go of its most basic language. Most if not all of the algorithmic approaches that have proved successful for solving temporal and numeric planning problems have originally been developed for the STRIPS language. If someone has a new idea for a planning algorithm or heuristic, he or she most certainly won’t implement it for PDDL2.1 level 3 in the first go. There is also the issue of accessibility of the competition, particularly to newcomers. We made a serious effort to make even

\*Institut für Informatik, Universität Freiburg, Germany

†Fachbereich Informatik, Universität Dortmund, Germany. Supported by DFG

‡T-Mobile, Germany

§Departamento de Informática, PUC Rio, Brazil. Supported by CNPq.

¶National ICT Australia & Computer Sciences Laboratory, The Australian National University, Canberra, Australia

||Institut für Informatik, Universität Freiburg, Germany

<sup>1</sup>We even thought of separating the domains into a set of “application” benchmarks and a set of “structurally characteristic” benchmarks. We gave up on the idea to not overly complicate the competition and its evaluation.

the STRIPS versions of the IPC-4 domains an interesting range of benchmarks. Instead of dropping the more interesting problem constraints, we *compiled* as much of the domain semantics as possible down into the STRIPS format. While in most cases this lead to rather unusual (fully grounded) encodings, we believe that the IPC-4 STRIPS benchmarks are structurally a lot more interesting than most of the previous STRIPS benchmarks.

In the rest of this extended abstract, we include a short description of each of the IPC-4 domains. We list the domains in alphabetical order, and close the article with a few concluding remarks.

## Airport

The *Airport* domain was developed by Jörg Hoffmann and Sebastian Trüg. It is a PDDL adaption of an application domain developed by Wolfgang Hatzack (Hatzack & Nebel 2001), dealing with the problem of controlling the ground traffic on an airport (in such a way that the summed up travel time of all airplanes is minimised).

The problem instances in Airport specify the topology of the airport, as well as the inbound (planes that need to go to a parking position) and outbound (planes that need to go to a runway) traffic. The main problem constraint is that planes must not endanger each other. Which means that no two planes can share the same airport segment, and that a plane with running engines “blocks” a set of segments behind it (where the blocked set depends on the size category of the plane). The available actions are to “pushback” (move a plane away backwards from a parking position), to “startup” the engines, to “move” between segments, to “park” (turning off the engines), and to “takeoff” (which amounts to removing the plane from the airport).

The Airport domain versions are *non-temporal*, *temporal*, *temporal-timewindows*, and *temporal-timewindows-compiled*. The first of these versions is, as the name suggests, non-durational PDDL. In the second version, actions take time (e.g. moving across a segment takes the length of the segment divided by the speed of the plane). In the third version, there are additional time windows during which certain segments must not be used – namely, segments that belong to a runway and time windows during which a plane is known to land on that runway. The time windows are modelled using timed initial literals. In the fourth domain version, the timed initial literals are compiled into artificial (temporal) PDDL constructs, in order to make the domain version accessible to more planners.

In none of the domain versions were we able to model the true optimisation criterion – minimising makespan means minimising the travel time of the latest plane, rather than the summed up travel time of all planes. The difficulty in modelling the real optimisation criterion lies in accessing the time spans during which a plane does nothing, i.e., stays on an airport segment waiting until some other plane got out of the way. If one uses an explicit “wait” action, then one needs to introduce a discretisation of time (in order to say how long the plane is supposed to wait). We considered introducing a special “current-time” variable into PDDL2.2, returning the time of its evaluation in the plan execution. But, in a

discussion with the IPC-4 organising committee, we decided against this language feature as it seemed problematic from an algorithmic point of view, and didn’t seem to be very relevant anywhere except in Airport.

In all the domain versions, the problem constraints are modelled using ADL, i.e., complex preconditions and conditional effects. We compiled the ADL encodings to STRIPS by grounding out most of the operator parameters (for each individual problem instance, yielding an instance-specific domain file). The resulting STRIPS encodings formed alternative *formulations* of the domain versions, i.e. within each domain version we let the competitors choose to either attack the ADL formulation or the STRIPS formulation. The data were then evaluated together, i.e. treated as if they were all obtained on the same encoding. We applied this concept of domain *versions* and domain *version formulations* in all the IPC-4 domains.<sup>2</sup>

The Airport example instances were generated by Sebastian Trüg, using an airport simulation tool, called *Astras*, by Wolfgang Hatzack. Five scaling airport topologies were designed, the simulator was run, and code was implemented that, during a simulation, put out the traffic situations at selected individual time spots as the PDDL problem instances. 50 traffic situations were generated, and put out in the format needed for each of the domain versions. The second largest of the five airport topologies corresponds to one half of Munich airport, MUC. The largest of the topologies corresponds directly to the full MUC airport.

## Pipesworld

The *Pipesworld* domain is a PDDL adaption of an application domain developed by Frederico Liporace and others (Milidui, dos Santos Liporace, & de Lucena 2003), dealing with complex problems that arise when transporting oil derivative products through a pipeline system. Note that, while there are many planning benchmarks dealing with variants of transportation problems, transporting oil derivatives through a pipeline system has a very different and characteristic kind of structure. The pipelines must be filled with liquid at all times, and if you push something into the pipe at one end, something possibly completely different comes out of it at the other end. Additional difficulties that have to be dealt with are, e.g., *interface restrictions* (different types of products that must not interface each other in a pipe), *tankage restrictions* in areas (i.e., limited storage capacity defined for each product in the places that the pipe segments connect), and *deadlines* on the arrival time of products. In the form used in IPC-4, the Pipesworld domain was developed by Frederico Liporace and Jörg Hoffmann. In all versions of the domain, the product amounts dealt with are discrete in the sense that we assume a smallest product unit, called “batch”. Of course, in reality the product amounts dealt with are rational numbers. Using such a numeric en-

<sup>2</sup>We are aware that encoding details can have a significant impact on system performance. On the other hand, we believe it is important to keep the number of distinction lines in the competition data – which is already high – as low as possible. Most current systems ground the operators out as a pre-process anyway.

coding in IPC-4 seemed completely infeasible due to complications in the modelling, and the expected capabilities of the participating planners.

The problem instances in Pipesworld specify the topology of the pipeline network, the initial positions for all the batches and the goal positions for some of the batches, and the additional constraints imposed – interface restrictions, tankage restrictions, and/or deadlines. A possible action is to “push” a batch from an area into a pipe segment, making the last batch in the pipe come out at the other end. Pipe segments are modelled in a directional fashion, and we also need the inverse “pop” action where a new batch is inserted at the far end of the pipe, and the first batch in the pipe comes out. In the actual PDDL encodings used, these actions are split in several ways, to ease the modelling of their semantics. The main difficulty is that the actions must keep track of the internal state of the pipe segment involved. We introduced special case actions for pipe segments of length 1 (i.e., 1 batch). For pipe segments containing more than 1 batch, we split the push (pop) action into a push-start (pop-start) and a push-end (pop-end) action. While there is in principle no problem with doing the necessary updates within a single action, such an action contains rather many parameters. In particular, 3 parameters ranging over batches are needed – the batch to be pushed (popped), the first batch inside the pipe segment, and the last batch inside the pipe segment. Thus such an action has at least  $n^3$  ground instances in the presence of  $n$  batches. We found that this made the domain completely infeasible for any planner that grounded out the actions. In the splitted encoding, each action takes at most two batch parameters.

The Pipesworld domain versions are *notankage-nontemporal*, *tankage-nontemporal*, *notankage-temporal*, *tankage-temporal*, *notankage-temporal-deadlines*, and *notankage-temporal-deadlines-compiled*. All versions include interface restrictions. The versions with “tankage” in their name include tankage restrictions. In the versions with “temporal” in their name, actions take (different amounts of) time. The motivation for the durative actions, from an operational point of view, is that each pipeline segment has a maximum flow rate, and thus the content of some segments may be moved faster than others. The versions with “deadlines” in their name include deadlines on the arrival of the goal batches. One of these versions models the deadlines using timed initial literals, in the other version (naturally, with “compiled” in its name) these literals are compiled into artificial (temporal) PDDL constructs. None of the encodings uses any ADL constructs, so of each version there is just one (STRIPS) formulation.

The Pipesworld example instances were generated by Frederico Liporace, in a process going from random generators to XML files to PDDL files.<sup>3</sup> Five scaling network topologies were designed. For the domain versions without tankage restrictions and deadlines, for each of the network topologies 10 scaling random instances were gener-

ated. (Within a network, the instances scaled in terms of the total number of batches and the number of batches with a goal location.) For the instances featuring tankage restrictions or deadlines, the generation process was more complicated because we wanted to make sure to obtain only solvable instances. For the tankage restriction examples, we ran Mips on the respective “notankage” instances, with incrementally growing tankage. We chose each instance at a random point between the first instance solved by Mips, and the maximum needed tankage (enough tankage in each area to accommodate all instance batches). Some instances could not be solved by Mips even when given several days of runtime, and for these we inserted the maximum tankage. For the deadline examples, we ran Mips on the corresponding instances without deadlines, then arranged the deadline for each goal batch at a random point in the interval between the arrival time of the batch in Mips’s plan, and the end time of Mips’s plan. The instances not solved by Mips were left out.

## Promela

*Promela* is the input language of the ACM awarded model checker SPIN (Holzmann 1997). It is designed to ease specification of asynchronous communication protocols, which are to be validated by SPIN for having no specification error. Otherwise the tool returns an error trail as a counterexample. A Promela model consists of a set of processes, and communication between them is performed via message queues or shared access to global variables. Each process can nondeterministically choose one of its transitions that fulfills the condition an optional guard imposes. The IPC-4 Promela domain was created by Stefan Edelkamp.

To allow STRIPS encodings for IPC-4, we selected two simple communication protocols: a solution for the *Dining Philosopher* problem, and the *Optical Telegraph* protocol. Both domains restrict to pure message passing, so that no shared access to global variables is used. The models are distributed together with our experimental model checking tool HSF-SPIN (Edelkamp, Leue, & Lluich-Lafuente 2004), that extends SPIN with heuristic search strategies to improve error detection. In both cases we used one scaling parameter, namely the number of philosophers and the number of control stations, respectively.

In order to generate problem instances fully automatically, we apply a compiler that transforms Promela specifications into PDDL2.2. The compilation process and an exposition for one of the protocols are described in (Edelkamp 2003). The compiler features some but not all static language constructs of Promela. Although not covered by the IPC-4 benchmark set, the work also showed that including communication via global variables and assignments of (not necessarily linear) arithmetic expressions to variables can be expressed in PDDL2.2. Besides deadlocks, violations to assertions and global invariants can also be converted into PDDL2.2 planning goals. For more complex error descriptions, e.g. liveness errors, temporally extended goals are needed. One of the core differences between Promela and PDDL2.2 expressiveness are dynamic processes. An according PDDL model would require a language extension

<sup>3</sup>The same XML file is mapped into different PDDL files depending on the kind of encoding used; there was a lot of trial and error before we came up with the final IPC-4 encoding.

for *dynamic object creation*. Fortunately, the core of most Promela specifications in our own collection is static.

Both protocols are known to contain deadlocks. In the PDDL2.2 descriptions, we utilised the finite state automata representation for the processes and communication queues that is inferred by SPIN. All active Promela processes are typed, enumerated and assigned to a unique object id. Each process consists of local states and transitions, with the queue read and write operations specifically tagged. In the PDDL model, a local state transition is first *activated* before according changes to the state variables or updates to the queue are executed. Finally the state change is *performed*. To ease parsing, state transitions use a reduced ASCII set.

Queues model communication channels, in which messages (and optional data) is written and read by the processes. The main idea in modelling queues is to represent arrays of size  $k$  in a ring structure: bucket 0 is the successor of bucket  $k-1$  with a head and a tail pointer that are moving. A queue is either empty or full if both pointers refer to the same queue state. As a special case the queues can consist of only one queue state, so the successor bucket of bucket 0 is the bucket itself. In this case the grounded propositional encoding includes operators with add and delete lists that share the same atom, so that we rely on the semantics of STRIPS, saying that deletion is done first.

If the message for reading does not match or the queue capacity is either too small or too large, the according local state transitions will block. If all active transitions in a process block, the process itself will block. If all processes are blocked, we have a deadlock in the system. Detection of a deadlock is crucial and is implemented either as a collection of PDDL2.1 actions or, more elegantly, as a set of PDDL2.2 derived predicates, automatically inferring that all processes for a state transition are blocked.

With each protocol we provide four different domain versions: *plain*, a purely propositional specification with specific actions that have to be applied to fix the deadlock; *fluents* an alternative to the above with numerical state variables that encodes the size of the queues and the messages used to access their contents; *derivedpredicates*, which contains derived predicates to infer deadlocks; and *fluents-derivedpredicates*, which is equivalent to *derivedpredicates* and uses fluents instead of propositions for encoding queue sizes and messages. We use one formulation that uses the ADL constructs *quantification*, *disjunctive* and *negated preconditions*; and one where the same semantics are compiled into pure (propositional) STRIPS. Unfortunately, the larger problem instances of these STRIPS formulations were too big to be stored on disk. We kept *fluent*-domains as separated *versions* instead of different *formulations* to compare pure propositional and numerical exploration efficiencies and to emphasise that numerical state variables are essential for more complex model checking domains.

## PSR

The *Power Supply Restoration (PSR)* domain is a PDDL adaptation of an application domain investigated by Thiébaux and others (Thiébaux *et al.* 1996; Thiébaux & Cordier 2001), which deals with reconfiguring a faulty

power distribution system to resupply customers affected by the faults. A power distribution system is viewed as a network of electric lines connected by switches and fed via a number of power sources. When a power source feeds a faulty line, the circuit-breaker fitted to this source opens to protect the rest of the network from overloads. This leaves *all* the lines fed by the source without power. The problem consists in planning a sequence of switching operations (opening or closing switches and circuit-breakers) bringing the network into a configuration where non-faulty lines are resupplied.

In the original PSR problem (Thiébaux & Cordier 2001), various numerical parameters such as breakdown costs and power margins need to be optimised, subject to power capacity constraints. Furthermore, the location of the faults and the current network configuration are only partially observable, which leads to a tradeoff between acting to resupply lines and acting to reduce uncertainty. In contrast, the version used for IPC-4 is set up as a pure goal-achievement problem (the goal specifies which lines must be (re)-supplied), numerical aspects are ignored, and total observability is assumed. The choice of leaving out the numerical aspects was motivated by the difficulty of encoding and solving even the basic problem. The IPC-4 PSR domain was developed by Sylvie Thiébaux and Jörg Hoffmann. We benefited from contributions by Piergiorgio Bertoli, Blai Bonet, Alessandro Cimatti, and John Slaney, some of which are reported in (Bertoli *et al.* 2002; Bonet & Thiébaux 2003).

PSR problem instances specify (1) the network topology, i.e., the objects in the network (the lines, the switches, the sources/circuit-breakers), and their connections, (2) the initial configuration, i.e., the initial positions (open/closed) of the switches and circuit-breakers, and (3) the modes (faulty or not) of the various lines. Among those, only the devices' positions can change. A number of other predicates are derived from these basic ones. They model the propagation of the current into the network with a view to determining which lines are currently fed and which sources are *affected* by a fault, i.e. feed a fault. The closed-world assumption semantics of PDDL2.2 derived predicates is exactly what is needed to elegantly encode such relations. These require a recursive traversal of the network paths which is naturally represented as the transitive closure of the connection relation of the network.

The goal in a problem instance asks that given lines be fed and all sources be unaffected.<sup>4</sup> The available actions are closing and opening a switch or a circuit-breaker. In addition, there is an action *wait*, which models the event of circuit-breakers opening when they become affected. *Wait* is applicable when an affected source exists, and is the only applicable action in that case. The goal and this together ensures that the *wait* action is applied as soon as a source is affected. The effect of the *wait* action is to open all the affected circuit-breakers. It would have been possible to encode the opening of affected breakers as a conditional effect

<sup>4</sup>Note that after the circuit-breaker of an affected source opens, this source is not affected any more, as it does not feed any line.

of the close action. However, this would have required more complex derived predicates with an additional device as parameter and a conditional flavor, specifying, e.g., whether or not a circuit-breaker *would be* affected *if* we were to close that device.

We use four domain versions of PSR in IPC-4. Primarily, these versions differ by the size of the problem instances encoded. The instance size determined in what languages we were able to formulate the domain version. We tried to generate instances of size appropriate to evaluate current planners, i.e., we scaled the instances from “push-over for everybody” to “impossibly hard for current automated planners”, were we got our intuitions by running a version of FF enhanced to deal with derived predicates. The largest instances are of the kind of size one typically encounters in the real world. More on the instance generation process below. The domain versions are named 1. *large*, 2. *middle*, 3. *middle-compiled*, and 4. *small*. Version 1 has the single formulation *adl-derivedpredicates*. Version 2 has the formulations *adl-derivedpredicates*, *simpleadl-derivedpredicates*, and *strips-derivedpredicates*. Version 3 has the single formulation *adl*, and version 4 has the single formulation *strips*. The formulation names simply give the language used. Version 1 contains the largest instances, versions 2 and 3 contain (the same) medium instances, and version 4 contains the smallest instances. The *adl-derivedpredicates* formulation is inspired from (Bonet & Thiébaux 2003), makes use of derived predicates as explained above, and of ADL constructs in the derived predicate, action, and goal definitions. In the *simpleadl-derivedpredicates* and *strips-derivedpredicates* formulations, all ADL constructs (except conditional effects in the *simpleadl* case) are compiled away using automated software (basically, FF’s pre-processor). The resulting encodings are fully grounded and significantly larger than the original, while on the other hand the length of plans remains completely unaffected. The pure *adl* formulation is obtained from the *adl-derivedpredicates* formulation by compiling derived predicates away using the method described in (Thiébaux, Hoffmann, & Nebel 2003). While there is no increase in the domain size, this compilation scheme can lead to an exponential increase in plan length in the worst case. For the PSR instances we generated, we observed only a polynomial blow up. Nevertheless we felt that this increase in plan length was too much to make for a useful direct comparison of data generated for *adl-derivedpredicates* as opposed to *adl*, and we separated the *adl* formulation out into domain version 3 as listed above.

The *strips* domain formulation proved quite a challenge. No matter how hard we tried, compiling both derived predicates and ADL constructs away led to either completely unmanageable domain descriptions or completely unmanageable plans. We therefore adopted a different fully-grounded encoding inspired from (Bertoli *et al.* 2002), which is generated from a description of the problem instance by a tool performing some of the reasoning devoted to the planner under the other domain versions. As a result, the STRIPS encoding is much simpler and only refers to the positions of the devices and not to the lines, faults, or connections. Also we were still only able to formulate comparatively small in-

stances in STRIPS, without a prohibitive blow-up in the encoding size.

The PSR instances were randomly generated using John Slaney’s randomnet program. Power distribution networks often have a meshable structure exploited radially: the path taken by the power of each source forms a tree whose nodes are switches and whose arcs are electric lines; terminal switches connect the various trees together. Randomnet takes as input the number of sources, a percentage of faulty lines, and a range of parameters for controlling tree depth, branching, and tree adjacency, whose default values are representative of real networks. Randomnet randomly selects a network topology and a set of faulty lines. These are turned into the various PDDL encodings above by a tool called net2pddl,<sup>5</sup> implemented by Piergiorgio Bertoli and Sylvie Thiébaux. The instances we generated make use of randomnet default settings, except for the maximal depth of trees which takes a range of values up to twice the default, leading to harder problems. The percentage of faulty lines ranges from 0.1 to 0.7.

## Satellite

The *Satellite* domain was introduced in IPC-3 by Derek Long and Maria Fox (2003). It is motivated by a NASA space application: a number of satellites has to take images of a number of spatial phenomena, obeying constraints such as data storage space and fuel usage. In IPC-3, there were 5 versions of the domain, corresponding to different levels of the language PDDL2.1: *Strips*, *Numeric*, *SimpleTime* (action durations are constants), *Time* (action durations are expressions in static variables), and *Complex* (durations and numerics, i.e. the “union” of Numeric and Time).

The adaption of the Satellite domain for IPC-4 was done by Jörg Hoffmann. All IPC-3 domain versions and example instances were re-used, except SimpleTime – like in the other IPC-4 domains, we didn’t want to introduce an extra version distinction just for the difference between constant durations and static durations. On top of the IPC-3 versions, 4 new domain versions were added. The idea was to make the domain more realistic by additionally introducing time windows for the sending of the image data to earth, i.e. to antennas that are visible for satellites only during certain periods of time – according to Derek Long, the lack of such time windows was the main shortcoming of the IPC-3 domain.

We extended the IPC-3 Time domain version to two IPC-4 domain versions, *Time-timewindows* and *Time-timewindows-compiled*. We extended the IPC-3 Complex domain version to the two IPC-4 domain versions *Complex-timewindows* and *Complex-timewindows-compiled*. In all cases, we introduced a new action for the sending of data to an antenna. An antenna can receive data of only a single satellite at a time, an antenna is visible for only subsets of the satellites for certain time periods, and the sending of

<sup>5</sup>Randomnet and net2pddl are available from the PSR benchmark resource web page <http://csl.anu.edu.au/~thiebaux/benchmarks/pds>, along with various other tools and papers of interest.

an image takes time proportional to the size of the image. The time windows were modelled using timed initial literals, and in the “-compiled” domain versions, these literals were compiled into artificial PDDL constructs. None of the domain versions uses ADL constructs, so of all versions there is only a single (STRIPS) formulation.

The instances were generated as follows. Our objectives were to clearly demonstrate the effect of additional time windows, and to produce solvable instances only. To accomplish the former, we re-used the IPC-3 instances, so that the only difference between, e.g., Time and Time-timewindows, lies in the additional time window constructs. To ensure solvability, we implemented a tool that read the plans produced by one of the IPC-3 participants, and then arranged the time windows so that the input plan was suitable to solve the enriched instance. It is important to note here that the time windows were *not* arranged to exactly meet the times extracted from the IPC-3 plan. Rather, we introduced one time window per each 5 “take-image” actions, made the antenna visible during that time window for only the respective 5 satellites, and let the image sizes be random values within a certain range where the time window was 5 times as long as the sending time resulting from the maximum possible size.

Of course, the above generation process is arranged rather arbitrarily, and the resulting instances might be a long way away from the typical characteristics of the Satellite problem as it occurs in the real world. While this isn’t nice, it is the best we could do without inside knowledge of the application domain, and it has the advantage that the enriched instances are solvable, and directly comparable to the IPC-3 ones.

In the new domain versions derived from Complex, we also introduced utilities for the time window inside which an image is sent to earth. For each image, the utility is either the same for all windows, or it decreases monotonically with the start time of the window, or it is random within a certain interval. Each image was put randomly into one of these classes, and the optimisation requirement is to minimise a linear combination of makespan, fuel usage, and summed up negated image utility.

## Settlers

The *Settlers* domain was introduced in IPC-3 by Derek Long and Maria Fox (2003). It makes extensive use of numeric variables. These variables carry most of the domain semantics, which is about building up an infrastructure in an unsettled area, involving the building of housing, railway tracks, sawmills, etc. The domain was included into IPC-4 in order to pose a challenge for the numeric planners – the other domains mostly do not make much use of numeric variables, other than computing the (static) durations of actions. We used the exact same domain file and example instances as in IPC-3, except that we removed some universally quantified preconditions to improve accessibility for planners. The quantifiers ranged over domain constants only so they could easily be replaced by conjunctions of atoms.

## UMTS

The *UMTS* domain has been developed by Roman Englert (2003). It enables the execution of several (data) applications in mobile terminals. To start an application in a mobile terminal the UMTS call set-up is required. This procedure takes between a couple of seconds for an interactive game like chess and 30 seconds for WAP access. Often users start several applications and as a consequence the waiting period until the call set-ups are executed takes several minutes. Therefore, optimisation of the UMTS call set-up is needed, where each application call is partitioned into modules (Englert 2005). The call set-up via software agents consists of eight discrete modules:

- terminal resource management (*trm*): an application start follows the resource availability check in the mobile terminal and the resource allocation
- connection timing (*ct*): connection set-up duration is monitored in the bearer and in case of failure feedback to the terminal is given (within a certain time, e.g. 1 sec.)
- agent management (*am*): requirements of mobile applications are transferred to bearer, e.g. Quality of service (QoS), required data volume, ...
- agent execution environment mobile (*aeem*): information about mobile application are sent to *am*, e.g. required servers, ...
- radio resource control (*rrc*): allocation of QoS by logical resources
- radio access bearer: (*rab*) bearer allocation of QoS and in case of failure initiation of resource negotiation with mobile terminal
- agent execution environment internet (*aeei*): data transfer for application set-up from mobile terminal to core network and PDN, and vice versa
- bearer service (*bs*): bearer establishment and feedback to mobile application,

To start the execution of a mobile application the modules are executed in sequential order. If several applications are initiated, some modules can be executed in parallel. The modules obey the following partial execution order: *trm* before *ct*, *ct* before *rrc* and *am*, *am* before *aeem*, *aeem* and *rrc* before *rab*, *rab* before *aeei*, *aeei* before *bs*, with *bs* being final. A detailed documentation on UMTS can be found in (Holma & Toskala 2000).

The PDDL2.2 translation of UMTS was established by Stefan Edelkamp and Roman Englert. Actions were attached to execution time, calling for Level 3 temporal planning. Instances are scaled to setup 1 up to 10 applications, a range that is practically motivated. Compared to other benchmarks, problem and domain description are comparable small to rise a challenge especially for optimal temporal planning approaches. However, real-time is required for practical purposes. Action durations are given in milliseconds and are selected due to practical constraints. The entire benchmark set was completed by running a problem generator that performs a realistic perturbation on the action execution times.

In the form used in IPC-4, the UMTS domain has six versions. The first three are: *temporal*, a domain version with no timing constraints, *temporal-timewindows*, a domain version with PDDL2.2 timed initial facts, and *temporal-timewindows-compiled*, a domain version with a PDDL2.1 wrapper encoding for the timed initial literals. The second domain version set *flaw-temporal*, *flaw-temporal-timewindows*, and *flaw-temporal-timewindows-compiled*, includes an additional but practical motivated *flaw* action that can affect plan finding, since it offers a shortcut to a relaxed plan not needed for a valid one, and, in order to determine that this action is not required, negative interactions have to be computed.

All domain versions have one formulation, namely *strips-fluents-temporal*, where numerical fluents, but - except typing - no ADL constructs are used. In all instances, the plan objective is to minimise *makespan*. The *temporal* and *temporal-timewindow* problem specifications were tested with the MIPS planner (Edelkamp 2004).

Besides action duration, the domain encodes scheduling types of resources, consuming some amount at action initialisation time and releasing the same amount at action ending time. Renewable global resources have not been used in planning benchmarks before, and the good news are that PDDL2.2 is capable of expressing them. In fact we used a similar encoding to the one that we found for *Job-* and *Flow-Shop* problems. As one feature, actions are defined to temporarily produce rather than to temporarily consume resources. As PDDL2.2 has no way of stating such resource constraints explicitly, planners that want to exploit that knowledge have to look for a certain patterns of *increase/decrease* effects to recognise them.

In UMTS, two actions can both check and update the value of some resources (e.g. *has-mobile-cpu*) at their starting (resp. ending) time points as far as the start (resp. ending) events are separated by  $\epsilon$  time steps, where  $\epsilon$  is minimum slack time required between two dependent events. We first thought about modelling renewable resources with an *over all* construct. But in this case, the invariant condition of the action has to check, what the *at start* event did change. We decided that this is not the best choice for a proper durative action. Consequently, the durative actions require that there is enough of the resource available *before* adding the amount used.

The domain assumes that the mobile applications run on one mobile terminal. However, they can also be distributed on to several mobile terminals. Additionally, the resource modeling of the UMTS network is constrained to the most important parameters (in total 15). In real networks several hundred parameters are applied.

## Concluding Remarks

In a field of research about general reasoning mechanisms, such as AI planning, it is essential to have appropriate benchmarks – benchmarks that reflect possible applications of the developed technology, and that help drive research into new and fruitful directions. In the development of the benchmark domains and instances for IPC-4, the authors have invested significant effort into creating such a set of

appropriate benchmarks for AI planning. The domains are mostly still far away from “real-world” problems, and we are aware that, e.g., fully grounded STRIPS encodings aren’t nice and pose a serious problem for systems that don’t use the standard pre-processes. Nevertheless we believe that the IPC-4 domains constitute a significant step into the right direction, and that they form an interesting range of benchmarks. We hope they will become standard benchmarks in the coming years.

**Acknowledgements.** We would like to thank the competitors for their detailed comments about found bugs in our domains, and we would like to thank Malte Helmert for various useful tools that helped remove some of these bugs.

## References

- Bertoli, P.; Cimatti, A.; Slaney, J.; and Thiébaux, S. 2002. Solving power supply restoration problems with planning via symbolic model-checking. In *Proc. 15<sup>th</sup> European Conference on Artificial Intelligence (ECAI-02)*, 576–580.
- Bonet, B., and Thiébaux, S. 2003. GPT meets PSR. In *13<sup>th</sup> International Conference on Automated Planning and Scheduling (ICAPS-03)*, 102–111.
- Edelkamp, S.; Leue, S.; and Lluch-Lafuente, A. 2004. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology*. To appear.
- Edelkamp, S. 2003. Promela planning. In *Workshop on Model Checking Software (SPIN)*, Lecture Notes in Computer Science, 197–212. Springer.
- Edelkamp, S. 2004. Extended critical paths in temporal planning. In *Proceedings ICAPS-Workshop on Integrating Planning Into Scheduling*.
- Englert, R. 2003. Re-scheduling with temporal and operational resources for the mobile execution of dynamic UMTS applications. In *KI-Workshop AI in Planning, Scheduling, Configuration and Design (PUK)*.
- Englert, R. 2005. Planning to optimize the umts call setup for the execution of mobile agents. *Journal of Applied Artificial Intelligence (AAI)*. To appear.
- Hatzack, W., and Nebel, B. 2001. The operational traffic control problem: Computational complexity and solutions. In Cesta, A., and Borrajo, D., eds., *Recent Advances in AI Planning. 6th European Conference on Planning (ECP’01)*, 49–60. Toledo, Spain: Springer-Verlag.
- Hoffmann, J. 2002. Local search topology in planning benchmarks: A theoretical analysis. In Ghallab, M.; Hertzberg, J.; and Traverso, P., eds., *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02)*, 92–100. Toulouse, France: Morgan Kaufmann.
- Holma, H., and Toskala, A. 2000. *WCDMA for UMTS - Radio Access for 3rd Generation Mobile Communications*. Wiley & Sons.
- Holzmann, G. J. 1997. The model checker Spin. *IEEE Trans. on Software Engineering* 23(5):279–295. Special issue on Formal Methods in Software Practice.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*. Special issue on the 3rd International Planning Competition, to appear.

Milidiu, R. L.; dos Santos Liporace, F.; and de Lucena, C. J. 2003. Pipesworld: Planning pipeline transportation of petroleum derivatives. In *Proceedings ICAPS-03 Workshop on the Competition*.

Thiébaux, S., and Cordier, M.-O. 2001. Supply restoration in power distribution systems — a benchmark for planning under uncertainty. In *Proc. 6th European Conference on Planning (ECP-01)*, 85–95.

Thiébaux, S.; Cordier, M.-O.; Jehl, O.; and Krivine, J.-P. 1996. Supply restoration in power distribution systems — a case study in integrating model-based diagnosis and repair planning. In *Proc. 12<sup>th</sup> Conference on Uncertainty in Artificial Intelligence (UAI-96)*, 525–532.

Thiébaux, S.; Hoffmann, J.; and Nebel, B. 2003. In defense of pddl axioms. In *18<sup>th</sup> International Joint Conference on Artificial Intelligence (IJCAI-03)*, 961–966.



# Macro-FF

**Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer**

Department of Computing Science, University of Alberta  
Edmonton, Alberta, Canada T6G 2E8  
{adib,emarkus,mmueller,jonathan}@cs.ualberta.ca

## Abstract

This document describes Macro-FF, an adaptive planning system developed on top of FF version 2.3. The original FF is a fully automatic planner that uses a heuristic search approach. In addition, Macro-FF can automatically learn and use macro-actions with the goal of reducing the number of expanded nodes in the search. Macro-FF also includes implementation enhancements for reducing space and CPU time requirements that could become performance bottlenecks in some problems.

## Introduction

Macro-FF is an extension of the automatic planner FF version 2.3 (Hoffmann & Nebel 2001). We developed a first version of Macro-FF as a tool for exploring how macro-actions can reduce the complexity of automated planning (Botea, Müller, & Schaeffer 2004). Further extensions have been implemented to prepare Macro-FF for participating in the fourth international planning competition (IPC4). Macro-FF is designed for classical planning and can use both STRIPS and ADL domain formulations. The plans that Macro-FF produces are not guaranteed to be optimal. The system has no capabilities for temporal and metric planning, and implements no support for derived predicates and timed initial literals.

This extended abstract summarizes the architecture of Macro-FF. The structure of our presentation is the following: First, we provide a brief description of FF, focusing on the parts that are relevant for our work. Next, we describe the main contributions that we have added to the original FF. The extensions that we present mainly go into two directions:

- Speeding up search with macro-operators. A macro-operator is an ordered sequence of operators together with a variable mapping showing how the variable sets of operators overlap. The intuition for using macro-actions is that several actions can often work in a sequence to accomplish a local task (e.g., first take the key out of the pocket, next unlock the door). Identifying and exploiting such sequences have a significant potential to reduce the overall planning effort. Macro-FF can automatically

learn and use macro-actions with the goal of reducing the number of expanded nodes in the search.

- Implementation enhancements for reducing memory and CPU time requirements. The number of expanded nodes and the solution quality are not affected by changes in this category. However, when the memory or CPU time necessary to solve a problem are larger than the available resources, this kind of improvements can make the difference between failure and success in solving a problem.

## Overview of FF

FF is a state-of-the-art fully automatic planner that uses a heuristic search approach. The basic version of FF, which we started from, is designed for classical planning. Specialized versions of FF have capabilities for planning with numerical state variables (Metric-FF) and planning with incomplete information (Conformant-FF).

FF uses a preprocessing phase that includes the generation of all facts (i.e., instantiated predicates) and actions (i.e., instantiated operators) that could possibly be used in the current problem instance. These elements, which are extensively used during the search, become available at little runtime cost.

FF automatically computes a heuristic state evaluator that guides the search process. Given a state, the distance to a goal state is approximated by the length of a relaxed plan that achieves the goal conditions starting from the current state. This plan is computed in a relaxed GRAPHPLAN framework, where the delete effects of actions are ignored.

The planner implements two search algorithms. Enforced hill climbing (EHC) is a fast but incomplete algorithm that greedily searches for a goal state in the problem space. EHC starts from the initial state and performs a local search using a breadth-first strategy. When a state with a better evaluation than the starting state is found, the current local search stops and a new local search is launched starting from the newly found state.

In EHC, the GRAPHPLAN computation for a state is used not only to find a heuristic evaluation, but also to further prune the search space through a mechanism called helpful action pruning. When a state is expanded, only moves that occur in the relaxed plan and belong to level 0 of the GRAPHPLAN (i.e., can be applied to the current

state) are considered. With no helpful action pruning, EHC is complete in undirected search spaces.

EHC stops when either a goal state is found, or the open list associated with the current local search is empty. When the second alternative occurs (i.e., EHC fails because of its incompleteness), a complete best-first search (BFS) algorithm is launched to find a path to a goal state.

## Learning and Using Macro-Operators

When treated as single moves, macro-actions have the potential of influencing the planning process in two important ways. First, macros can change the search space, adding to a node successor list states that would normally be achieved in several steps. Intermediate states in the macro sequence do not have to be evaluated, reducing the search costs considerably. In effect, the maximal depth of a search could be reduced for the price of slightly increasing the branching factor. Second, macros can improve the heuristic evaluation of states. As shown before, FF computes this heuristic by solving a relaxed planning problem (i.e., the delete effects of actions are ignored) in a GRAPHPLAN framework. Consider two normal actions that occur in a sequence in a relaxed plan. It is not guaranteed that this chaining translates to a valid action sequence in the real world (e.g., when the first action has a delete effect that is a precondition for the second action). Consider now the case when two actions compose a macro, so that the relaxed plan contains that macro rather than two separate actions. A relaxed macro can always be translated to its correspondent in the real world, as any other action does.

### Learning Phase

Macro-FF learns a set of macros through a training phase that uses several sample problems of a domain. Each training problem is first solved with no macros in use. The found plan  $P$  is represented as a directed *solution graph*, where each node represents a plan action, and edges show the relative order and distance between two actions in the solution. If action  $a_1$  occurs before action  $a_2$  in  $P$ , then a weighted edge  $e = (a_1, a_2)$  is added to the graph. The weight is the distance between  $a_1$  and  $a_2$  in the solution.

We define a macro-action as a linear sequence in the solution graph, with the corresponding parameter mapping. To reduce the training effort, our implementation considers only sequences of two consecutive actions as possible macros (i.e., only pairs of nodes linked by edges with weight 1).

The macro-actions are mapped to macro-operators by replacing the instantiated parameters with generic variables. Macro-operators have weights (initially set to 1.0) and are stored in a global list ordered by their weights.

For each macro-operator  $m$ , the current training problem is re-solved using  $m$ . To measure the usefulness of  $m$ , we compare the effort to solve the problem with macro  $m$  in use to the initial solving effort. We evaluate the effort to solve a problem as the total number of expanded nodes. The weight update formula for  $m$  uses the difference between  $N$  (the effort for solving the problem with no macros in use) and

$N_m$  (the effort when macro  $m$  is used). A sigmoid function maps the difference into the range  $(-1, 1)$ . The update value further contains the initial solution length as a multiplicative factor, which measures how hard the current problem is. The harder the problem, the larger this weight update should be. After the training phase completes, the best macros can be used in the solving phase.

### Solving Phase

**Current Implementation.** For IPC4, we store the macros using a *compact* representation. This includes the ids of the operators that compose the macro and the variable mapping, but ignores the precondition and effect formulas. In the solving mode, the compact patterns of the best macros are used for online checking if two instantiated actions compose a macro. The current implementation uses macros to change the search space (as shown next), but does not affect the computation of the heuristic state evaluation. Improving the heuristic state evaluation with macros is an important topic for future work.

To explore the search space more efficiently, we exploit the relaxed plan that the system computes for the current state to be expanded. Our idea is to try to execute parts of the relaxed plan in the real world, hoping to move toward a goal state faster. We examine the relaxed plan to find action sequences that match a macro pattern. Each time when such a sequence is identified, we check if this could be executed in the real world, starting from the current state. This verification is fast, as we do not compute the evaluation of the states along the execution path. If executing a macro-action succeeds, we consider the resulting state as a successor of the current state and add it to the open queue.

In enforced hill climbing, we order these macro successors before the regular successors of a state. In effect, macro successors are expanded earlier than regular successors. In addition, our code includes an ordering scheme for normal successors, which we had developed before using macro successors. In the current implementation, this still might be useful in cases when a macro is not part of the relaxed plan, but could occur in the real world. We order the normal successors giving priority to moves that continue as a macro sequence the last action on the current branch (i.e., the action that led to the currently expanded state  $S$ ). We split the normal successors of state  $S$  into two subsets  $\text{Succ}_1(S)$  and  $\text{Succ}_2(S)$ . Assume  $a_S$  is the action that we applied to obtain  $S$ , and  $a_{S'}$  is the action that we apply from  $S$  to obtain a successor  $S'$ . If pair  $(a_S, a_{S'})$  matches the pattern of a learned macro operator, then  $S' \in \text{Succ}_1(S)$ . Otherwise,  $S' \in \text{Succ}_2(S)$ . Elements from  $\text{Succ}_1(S)$  are ordered before elements from  $\text{Succ}_2(S)$ . Inside such a set, an additional move ordering scheme, preserved from the original FF, is applied.

In best-first search, macros act as a method for search depth control. In the original implementation, when a node is expanded, all its normal successors are added to the open list, except for states that have been visited before (a transposition table is used to identify duplicates). In addition to this, our new implementation explores branches that compose a macro more deeply. States are further expanded on

the branches that match a macro pattern, and the resulting states are added to the open list earlier than in the original FF.

**Alternative Approach.** Another possible way of using macro-operators is to add them as normal single-step operators to the initial domain formulation, as described in (Botea, Müller, & Schaeffer 2004). In this way, macro-actions are naturally used in both exploring the search space (i.e., as possible moves when nodes are expanded) and computing the heuristic state evaluation in the relaxed GRAPHPLAN framework, with no need to change the original code of FF. In effect, the number of expanded nodes can be reduced for the price of increased preprocessing time and cost per node at run-time.

This approach was hard to use in IPC4, as the macro-operators added to the domain formulation have to have complete PDDL definitions, including precondition and effect formulas. Expressing these formulas starting from the contained operators is easy in STRIPS, but hard in more complex PDDL subsets such as ADL, where the preconditions and the effects of the contained operators can interact in a very complex way. However, for IPC4, we used the ADL formulation for several domains that were available both in ADL and STRIPS. The reason is that the STRIPS formulation of these domains have a separate operator file for each problem. This makes our learning algorithm hard to apply, as several training problems are necessary for a given domain definition.

## Implementation Enhancements

The enhancements described in this section have the goal of reducing the space and CPU requirements of the planner, and do not affect the number of expanded nodes and the quality of found plans. We describe two enhancements, one for speeding-up the best-first search and one for reducing the space needs for the preprocessing.

The best-first search (BFS) algorithm uses an open list of nodes that have been generated but not expanded yet. The elements in this list are stored in increasing order according to their heuristic evaluation, so that the next node chosen for expansion is the most promising in the list. FF version 2.3 implements the open queue as a linear linked list. A node insertion requires a linear traversal of the list, so that the ordering of the list is preserved. Experiments with some of the competition problems have shown that this linear traversal can be a serious bottleneck for best-first search. We changed the original linked list of nodes to a linked list of buckets, where each bucket is a linked list of nodes having the same heuristic value. The insertion of a node requires finding the appropriate bucket for that node, which takes time linear in the number of different heuristic values in the open queue plus a constant time for inserting the node at the end of the bucket (this preserves the existing tie-breaking rule).

FF version 2.3 is optimized for speed by using preprocessing to a large extent. Some of the data structures used for holding the preprocessing information grow exponentially with the problem complexity, so that this method does not scale to more complex problems. We took an initial step to

address this problem by replacing a large lookup table by a different data structure. The lookup table was used for holding instantiated facts that occur in the initial state. The new implementation uses a balanced binary tree for logarithmic lookup time.

## Acknowledgment

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE). We thank Jörg Hoffmann for making the source code of FF available.

## References

- Botea, A.; Müller, M.; and Schaeffer, J. 2004. Using Component Abstraction for Automatic Generation of Macro-Actions. In *Proceedings of the International Conference on Automated Planning and Scheduling ICAPS-04*.
- Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14:253–302.

# Optiplan: Unifying IP-based and Graph-based Planning

Menkes van den Briel<sup>1</sup> and Subbarao Kambhampati<sup>2</sup>

<sup>1</sup>Department of Industrial Engineering, <sup>2</sup>Department of Computer Science and Engineering  
Arizona State University, Tempe AZ

## Abstract

The Optiplan planning system combines the ideas presented by Vossen *et al.* (1999) and Kautz and Selman (1998). It unifies integer programming with graph-based planning and computes optimal parallel length plans for STRIPS based planning problems. In addition, given a feasible parallel length, Optiplan can be used to minimize the number of actions, minimize action cost, or optimize any other objective that can be expressed as a linear function.

## OptiPlan

Optiplan is a domain independent planner that, like ILPLAN (Kautz & Walser 1999) and the “state change model” (Vossen *et al.* 1999), uses integer programming (IP) to solve STRIPS planning problems. The architecture of Optiplan is very similar to that of Blackbox (Kautz & Selman 1999) and GP-CSP (Do & Kambhampati 2001), but instead of unifying satisfiability or CSP with graph based planning, Optiplan uses integer programming. Like Blackbox and GP-CSP, Optiplan works in two phases. In the first phase the planning graph is build and transformed into an IP, then in the second phase the IP is solved using the commercial solver CPLEX (ILO 2002). The IP formulation is based on the state change formulation (Vossen *et al.* 1999), however, a few changes have been added that “strengthen” the original formulation and make it more general at the same time.

A practical difference between the state change model and Optiplan is that the former takes as input all ground actions and fluents over all time steps, while the latter takes as input just those actions and fluents that are instantiated by Graphplan (Blum & Furst 1995). The use of a planning graph has a significant effect on the size of the final encoding, independent of which combinatorial transformation method (IP, SAT, or CSP) is used. For example, Kautz and Selman (1999) pointed out that Blackbox’s success over Satplan was mainly explained by Graphplan’s ability to produce better, more refined, propositional structures than Satplan. Another, although minor, practical difference between Optiplan and the state change model is that Optiplan reads in pddl files, allowing it to be directly compared to other STRIPS based planners.

In order to present the improved state change formulation that is used in Optiplan we introduce the following sets and variables: (The reader familiar with the work by Vossen *et*

*al.* (1999) may want to skim through the formulation of the model and note that the variables  $x_{f,i}^{pre\,del}$ , for all  $f \in F, i \in 1, \dots, t$  have been deleted and the variables  $x_{f,i}^{del}$ , for all  $f \in F, i \in 1, \dots, t$  have been added to the original formulation.):

- $F$ , set of *fluents*, the set of all instantiated propositions;
- $A$ , set of *actions*, the set of all instantiated operators;
- $I \subseteq F$ , set of fluents that are true in the initial state;
- $G \subseteq F$ , set of fluents that must be true in the goal state;
- $pre_f \subseteq A, \forall f \in F$ , set of actions that have fluent  $f$  as precondition;
- $add_f \subseteq A, \forall f \in F$ , set of actions that have fluent  $f$  as add effect;
- $del_f \subseteq A, \forall f \in F$ , set of actions that have fluent  $f$  as delete effect;

The state change formulation defines variables for each step  $i$  in the planning graph. There are variables for the actions and there are variables for the possible state changes a fluent can make. For all  $a \in A, i \in 1, \dots, t$  we have the action variables

$$y_{a,i} = \begin{cases} 1 & \text{if action } a \text{ is executed in period } i, \\ 0 & \text{otherwise.} \end{cases}$$

The “no-op” actions are not included in the  $y_{a,i}$  variables but are represented separately by the state change variable  $x_{f,i}^{maintain}$ . For all  $f \in F, i \in 1, \dots, t$  we have the state change variables

$$x_{f,i}^{maintain} = \begin{cases} 1 & \text{if fluent } f \text{ is propagated in period } i, \\ 0 & \text{otherwise.} \end{cases}$$

$$x_{f,i}^{pre\,add} = \begin{cases} 1 & \text{if action } a \text{ is executed in period } i \\ & \text{such that } a \in pre_f \cap a \notin del_f, \\ 0 & \text{otherwise.} \end{cases}$$

$$x_{f,i}^{add} = \begin{cases} 1 & \text{if action } a \text{ is executed in period } i \\ & \text{such that } a \notin pre_f \cap a \in add_f, \\ 0 & \text{otherwise.} \end{cases}$$

$$x_{f,i}^{del} = \begin{cases} 1 & \text{if action } a \text{ is executed in period } i \\ & \text{such that } a \notin pre_f \cap a \in del_f, \\ 0 & \text{otherwise.} \end{cases}$$

In summary:  $x_{f,i}^{maintain} = 1$  if the truth value of a fluent is propagated;  $x_{f,i}^{preadd} = 1$  if an action is executed that requires a fluent and does not delete it;  $x_{f,i}^{add} = 1$  if an action is executed that does not require a fluent and adds it; and  $x_{f,i}^{del} = 1$  if an action is executed that does not require a fluent and deletes it.

There are a few differences with the original state change formulation and the formulation in Optiplan. Optiplan introduces the  $x_{f,i}^{del}$  variables in order to deal with actions that delete fluents without requiring them as preconditions. Many planning domains in the International Planning Competition 2004 have such actions, making the original state change formulation ineffective. In addition, the new formulation has substituted out all  $x_{f,i}^{preadd}$  variables by the expression  $\sum_{a \in pre_f \cup del_f} y_{a,i}$ . The updated formulation is given by:

$$\min \sum_{a \in A} \sum_{i \in T} y_{a,i} \quad (1)$$

$$\text{s. t. } x_{f,0}^{add} = 1, \forall f \in I \quad (2)$$

$$x_{f,0}^{add} = 0, \forall f \notin I \quad (3)$$

$$x_{f,t}^{add} + x_{f,t}^{maintain} + x_{f,t}^{preadd} \geq 1 \quad (4)$$

$$\sum_{a \in add_f / pre_f} y_{a,i} \geq x_{f,i}^{add} \quad (5)$$

$$y_{a,i} \leq x_{f,i}^{add} \quad (6)$$

$$\sum_{a \in pre_f / del_f} y_{a,i} \geq x_{f,i}^{preadd} \quad (7)$$

$$y_{a,i} \leq x_{f,i}^{preadd} \quad (8)$$

$$\sum_{a \in del_f / pre_f} y_{a,i} \geq x_{f,i}^{del} \quad (9)$$

$$y_{a,i} \leq x_{f,i}^{del} \quad (10)$$

$$x_{f,i}^{add} + x_{f,i}^{maintain} + x_{f,i}^{del} + \sum_{a \in pre_f \cup del_f} y_{a,i} \leq 1 \quad (11)$$

$$x_{f,i}^{preadd} + x_{f,i}^{maintain} + x_{f,i}^{del} + \sum_{a \in pre_f \cup del_f} y_{a,i} \leq 1 \quad (12)$$

$$x_{f,i}^{preadd} + x_{f,i}^{maintain} + \sum_{a \in pre_f \cup del_f} y_{a,i} \leq x_{f,i-1}^{preadd} + x_{f,i-1}^{add} + x_{f,i-1}^{maintain} \quad (13)$$

$$x_{f,i}^{preadd}, x_{f,i}^{add}, x_{f,i}^{del}, x_{f,i}^{maintain} \in \{0, 1\} \quad (14)$$

$$y_{a,i} \in \{0, 1\} \quad (15)$$

Where constraints (2), and (3) represent the initial state constraints, and (4) represent the goal state constraints. For all  $f \in \mathcal{F}, i \in 1, \dots, t$ , constraints (5) to (10) represent the logical interpretations between the action and state change

variables, and for all  $f \in \mathcal{F}, i \in 1, \dots, t$  constraints (11) and (12) make sure that fluents can only be propagated at period  $i$  if and only if there is no action in period  $i$  that adds or deletes the fluent. For all  $f \in \mathcal{F}, i \in 1, \dots, t$ , constraints (13) describe the backward chaining requirements. Constraints (14) and (15) are the binary constraints for the state change and action variables respectively. Since the constraints guarantee plan feasibility, no objective function is required, however, Optiplan uses an objective that minimizes the number of actions taken to guide the search.

Optiplan shows an increased performance over the original state change encoding, but it remains significantly slower than, for example, Blackbox(Chaff). Table 1 shows a comparison between the original state change formulation and Optiplan on a set of problems that we could test both encodings on. All tests were run on a Pentium 2.67 GHz with 1.00 GB of RAM and the IP encodings were solved using CPLEX 8.1. For all problems Optiplan creates smaller encodings than the original state change formulation, and in all but two instances (the two rocket problems) Optiplan's formulation is solved at least as fast as the original state change formulation.

Often times only a few nodes are explored in the branch-and-bound tree, this indicates that the LP relaxation provides a good approximation to the convex hull of integer solutions. Still, however, our IP approaches are easily outperformed by planners like Blackbox(Chaff). Possible reasons for this performance gap is that the CPLEX's integer programming solver is not specialized in solving pure 0-1 programming problems and because many "expensive" matrix operations are required when solving the LP relaxation. When these shortcomings are resolved, for example, through the use of special purpose algorithms like branch-and-cut, decomposition, or column generation, Optiplan and IP approaches in general could become competitive with other successful planners.

## References

- Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1636–1642.
- Do, M., and Kambhampati, S. 2001. Planning as constraint satisfaction: Solving the planning graph by compiling it into csp. *Artificial Intelligence* 132(2):151–182.
- ILOG Inc, Mountain View, CA. 2002. *ILOG CPLEX 8.0 User's Manual*.
- Kautz, H., and Selman, B. 1999. Blackbox: Unifying sat-based and graph-based planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 318–325.
- Kautz, H., and Walser, J. 1999. State-space planning by integer optimization. In *Proceedings of the 17th National Conference of the American Association for Artificial Intelligence*, 526–533.
- Vossen, T.; Ball, M.; Lotem, A.; and Nau, D. 1999. On the use of integer programming models in ai planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 304–309.

Problem	State change model				Optiplan			
	#Var.	#Cons.	#Nodes	Time	#Var.	#Cons.	#Nodes	Time
bw-sussman	196	347	0	0.01	105	142	0	0.01
bw-12step	1721	3163	15	4.53	868	1040	4	1.58
bw-large-a	2729	5106	0	5.04	1800	2104	0	3.91
bw-large-b	6502	12224	25	932.26	4780	5466	9	236.45
att-log0	33	41	0	0.01	6	8	0	0.01
att-log1	151	188	0	0.01	49	71	0	0.01
att-log2	330	420	14	0.05	130	193	0	0.01
att-log3	2334	3785	0	0.26	250	455	0	0.06
att-log4	2330	3775	42	0.59	449	850	0	0.12
att-loga	3146	5091	3583	366.44	1671	3258	80	29.84
rocket-a	1615	2694	169	8.80	1127	2365	49	12.38
rocket-b	1696	2829	122	8.27	1187	2516	27	11.58
log-easy	1521	2254	32	0.86	555	1088	0	0.14
log-a	3933	6306	174	48.36	1671	3258	80	29.74
log-b	4684	7202	1797	391.75	1962	3830	41	40.67
log-c	5886	9324	1378	946.23	2691	5370	114	183.96

Table 1: Comparing the original state change formulation with Optiplan. #Var. and #Cons. give the number of variables and constraints after CPLEX's presolve. #Nodes give the number of nodes that were explored during branch-and-bound before finding the first feasible solution.

# FAP: Foward Anticipating Planner

Guy Camilleri and Joseph Zalaket

IRIT CCI-CSC,  
Université Paul Sabatier,  
118 route de Narbonne,  
31062 Toulouse Cedex 4 FRANCE  
camiller@irit.fr zalaket@irit.fr

## Abstract

In this paper we introduce a new planning system FAP based on the heuristic search. For the heuristic calculation, FAP combines the techniques used in abstraction and heuristic planning. FAP calculates his heuristic by projecting the planning problem in a relaxed problem where the delete lists of the actions are ignored and the actions are grouped in sequences according to their order of application. FAP uses the calculated heuristic to guide its search on a N-Best-Search Hill-Climbing algorithm which is a combination of the N-Best-Search and Hill-Climbing algorithms.

## Introduction

The heuristic search has enhanced the performance of planning algorithms. Planners like HSP (0) HSPr (0) and FF (0) has shown the ability of solving large planning problems according to the classical previous planners. The heuristic used by the most of the current planners is based on the idea of McDermott (0) as well as Bonet et al. (0), which propose the relaxation of the problem in a simpler problem by ignoring the delete lists of the actions. Also the heuristic idea was early used in the hierarchical planning in a kind of relaxing the problem by projecting it in an abstract problem where the solution can be found faster (see planners like NOAH (0), NONLIN (0)). The abstraction used in hierarchical planning was often based on the actions or states grouping. In this paper we introduce a new planning system FAP based on the heuristic search. For the heuristic calculation, FAP combines the techniques used in abstraction and heuristic planning. FAP calculates the heuristic by projecting the planning problem in a relaxed problem where the delete lists of the actions are ignored and the actions are grouped in sequences according to their order of application. FAP uses the calculated heuristic to guide its search on a N-Best Hill-Climbing heuristic Search algorithm which is a combination of the N-Best heuristic Search and Hill-Climbing algorithms. In the rest of this paper we present an overview of our work. We explain the sequences meta-actions calculation. We present the generation of the sequences to finish with the main search algorithm.

## Overview

FAP is a forward planner in a state space which combines heuristic search planning techniques with a "state grouping" approach. As HSP (0), FF (0), etc. state's heuristic<sup>1</sup> is computed from a solution of a relaxed problem. The relaxed problem<sup>2</sup> ignores action's delete list and is solved through a planning graph similar to the GraphPlan's planning graph (0). The state grouping approach constitutes the main originality of this work. It aims at reducing state search space by grouping states, and is done through the generation of meta-actions "sequences" rather than building states shapes as in ShaPer (0) or states abstractions as in some hierarchical planning systems like ALPINE (0).

During the search, FAP generates new actions (or meta-actions) corresponding to the actions "sequence" called anticipations. These actions "sequence" are used like the other ones in the planning graph, in the states search space and can belong to other actions "sequence". In this way, all states are not considered in the search space.

All candidate actions to the sequence generation are pulled out from the planning graph. The actions selection during the extraction of the relaxed solution is essential because they do not only participate to the heuristic calculation but also to the state grouping. Currently, FAP extracts the relaxed solution in regression (from the last level) and uses some local criteria to select actions in the planning graph.

The main search algorithm used in FAP is an extension of the N-Best heuristic Search algorithm NBS (proposed in (0)) called N-Best heuristic Hill-Climbing Search algorithm NBHCS. This algorithm is complete and can be viewed as a kind of Hill-Climbing algorithm with a backtracking. Therefore FAP considers all applicable actions (not only the anticipations) to be complete.

For each state, FAP applies the following steps:

1. Relaxed planning graph building (similar to FF),

<sup>1</sup>The heuristic corresponds to an estimation of the distance in number of actions between the initial state and the goal.

<sup>2</sup>In STRIPS, a planning problem  $P = (O, I, G)$  is defined by a set of operators  $O$  which change the world state, an initial state  $I$  and a goal  $G$  to satisfy. The operators of the considered relaxed problem  $P' = (O', I, G)$  correspond to the operators of the problem  $P$  without the delete list.

2. Relaxed solution extraction which defines the candidate actions and the heuristic,
3. Sequence actions generation (in progression and then in regression).

In the first part of this paper, the meta-action "sequence" is briefly presented. Then, we expose the selection of relevant actions corresponding to the relaxed solution, the sequence actions generation and the state search algorithm NBHCS

### Meta-action "Sequence"

A ground action  $\alpha$  in STRIPS is described by the following lists:  $\text{param}(\alpha)$  is the list of action's parameters,  $\text{pre}(\alpha)$  is the list of preconditions which must hold for action's application,  $\text{add}(\alpha)$  and  $\text{del}(\alpha)$  lists are respectively the list of addition and the list of deletion of the action.

**Definition 1** The meta-action "sequence"  $\triangleright(\alpha_1, \alpha_2)$  is defined by:

- $\text{param}(\triangleright(\alpha_1, \alpha_2)) = (\alpha_1, \alpha_2)$
- $\text{pre}(\triangleright(\alpha_1, \alpha_2)) = \text{pre}(\alpha_1) \cup (\text{pre}(\alpha_2) \setminus \text{add}(\alpha_1))$
- $\text{add}(\triangleright(\alpha_1, \alpha_2)) = [\text{add}(\alpha_2) \cup (\text{add}(\alpha_1) \setminus \text{del}(\alpha_2))] \setminus \text{pre}(\triangleright(\alpha_1, \alpha_2))$
- $\text{del}(\triangleright(\alpha_1, \alpha_2)) = [\text{del}(\alpha_2) \cup (\text{del}(\alpha_1) \setminus \text{add}(\alpha_2))] \cap \text{pre}(\triangleright(\alpha_1, \alpha_2))$

Moreover, Fap used the following properties on the meta-action "sequence":

**Definition 2** Two ground actions  $\alpha_1$  and  $\alpha_2$  are S-independent iff  $\text{pre}(\triangleright(\alpha_1, \alpha_2)) = \text{pre}(\triangleright(\alpha_2, \alpha_1))$ ,  $\text{add}(\triangleright(\alpha_1, \alpha_2)) = \text{add}(\triangleright(\alpha_2, \alpha_1))$  and  $\text{del}(\triangleright(\alpha_1, \alpha_2)) = \text{del}(\triangleright(\alpha_2, \alpha_1))$ .

**Definition 3** A sequence  $\triangleright$  is correct iff it exists a state  $s$  reachable from the initial state such as  $\triangleright$  is applicable in  $s$ .

### Relevant actions extraction and sequences generation

For each state FAP builds a relaxed planning to calculate the heuristic of that state. Actions are extracted from this planning graph in regression. The extraction process starts from the goals in the last level and go back to the first level 0. For each goal in the current level, only one action is selected from the previous level for sequence building according to some local criteria. The preconditions of the chosen actions are then added to the goal set and then the process goes back to the previous level until the first level is reached.

The local criteria use the following relation of authorization:

**Definition 4** An action  $\alpha_1$  authorizes  $\alpha_2$  iff  $\text{del}(\alpha_1) \cap \text{pre}(\alpha_2) = \emptyset$

**Definition 5** A sequence  $\triangleright(\alpha_1, \alpha_2)$  where  $\text{level}(\alpha_1) = i$  and  $\text{level}(\alpha_2) = i + 1$  is considered useful at a level  $i$  iff it exists an atom  $p$  such as  $\text{level}(p) = i$  and  $p \in \text{add}(\alpha_1) \cap \text{pre}(\alpha_2)$ .

The local criteria describe some selection rules between actions belonging to two successive action's levels. For each goals  $g$  at a level  $i$ , an action  $\alpha$  is chosen at the level  $i - 1$  if 1)  $g \in \text{add}(\alpha)$  and 2) for all actions  $\beta$  in the level  $i$  such as  $g \in \text{pre}(\beta)$ ,  $\alpha$  authorize  $\beta$  and  $\alpha$  minimize the difficulty of  $\triangleright(\alpha, \beta)$  which  $\text{difficulty}(a) = \sum_{p \in \text{pre}(a)} \text{level}(p)$ . From

this selection, only actions which maximize the number of goals of the level  $i$  are chosen so as all level goals belong to an add list of these actions.

The meta-actions "sequences" are generated from a partial planning graph containing only the extracted actions. A first generation is done in forward from the actions in the level 0 in the following way: if all actions  $\alpha_i$  in level 0 are many to many S-independent then generate the sequence  $\triangleright_i \alpha_i$ . Then for all generated sequences  $\triangleright_k$  in a level  $i$  and all actions  $\beta$  in the level  $i + 1$ , only the useful sequences  $\triangleright(\triangleright_k, \beta)$  are computed. The process stops when the last level is reached or if any sequences can be generated at the current level  $c$ .

In the second generation, only the useful sequences are computed by pairs of successive levels in backward from the last level to the level  $c$ .

### NBHCS algorithm

The search algorithm used in FAP is an instantiation of the N-Best heuristic Search Algorithm (NBS). The NBS algorithm is at a time a functional extension and a simpler implementation of the First Best Search algorithm. In many planning problems, a state has a big number of successors, which decreases the planning performance if all of them are visited. The idea of the NBS algorithm is to generate a limited number  $N$  of successors at a time instead of generating all of them, then to expand the graph for the next  $N$  successors if no solution found and so on. Moreover, because the graph can be expanded every time the solution is missed up to containing all the successors, this algorithm is complete.

In the figure 1, the NBS algorithm is presented. The search process could be defined as a quadruplet  $(S_c, G, \Gamma_c, S_i)$  where  $S_c$  is the current best state,  $G$  is the goal,  $\Gamma_c$  is the set of operators applicable to  $S_c$  and  $S_i$  is the initial state. Any state  $S_n$  is completely expanded when its successors states are memorized and this state is kept in a list of all completely expanded states named Closed. A state  $S_k$  is partially expanded whenever it does not have any memorized successor or a part of its successor states are kept, all of these states are included in a list of states called Open.

Our N-Best heuristic Hill-Climbing Search algorithm is an NBS algorithm with a specification of the generate\_best\_successors function (see figure 2). As in Hill-Climbing search algorithm, the process of generating successors stops when a best successor is found. Let remark that in FAP the order in which the actions are memorized in the set  $\Gamma_c$  is very important because it defines the expansion strategy. The memorized order is: the generated sequences, the helpful actions (like the ones used in FF) and then the others, of course all these actions are applicable in the current state. Therefore, in a first stage the sequences are ap-



plied, in a second the helpful and then the other actions. By this way, FAP is complete.

---

**Algorithm 1** The N-Best heuristic Search Algorithm

---

```

Open  $\leftarrow \{(S_i, \Gamma_0)\}$ ;
Closed  $\leftarrow \emptyset$ ;
while Open  $\neq \emptyset$  do
   $(S_c, \Gamma_c) = \text{get\_state\_with\_min\_f}(\text{Open})$ ;
  generate\_best\_successors(N,  $S_c, \Gamma_c$ );
  if best\_successors( $S_c$ )  $\cap G \neq \emptyset$  then
    return  $S_c$ ;
  end if
  Open  $\leftarrow \text{Open} \cup \text{best\_successors}(S_c)$ ;
  if  $\Gamma_c = \emptyset$  then
    Open  $\leftarrow \text{Open} \setminus \{(S_c, \Gamma_c)\}$ ;
    Closed  $\leftarrow \text{Closed} \cup S_c$ ;
  else
    update_ $\Gamma(S_c, \Gamma_c)$ ;
  end if
end while
return Failure;

```

---



---

**Algorithm 2** generate\\_best\\_successors strategy

---

```

Successors  $\leftarrow \emptyset$ ;
repeat
   $\gamma \leftarrow \text{element}(\Gamma_c)$ ;
   $S \leftarrow \text{apply}(S_c, \gamma)$ ;
  Successors  $\leftarrow \text{Successors} \cup \{S\}$ ;
until  $f(S) < f(S_c)$ 
return Successors;

```

---

## Conclusion

This paper shows a new heuristic search planner based on the problem relaxation by action grouping. In the palnnign-graph, the generation of sequences "actions group" and their application can be more informative as a heuristic guide than the separated actions application. The main search algorithm can recuperate the time that the computation of sequences takes. Therefore, the main search algorithm can use a shorter path to achieve the goal with sequences than with direct heuristics. The main stake is to build the best sequences by choosing the actions as possible in the order of their applications to access the result as fast as possible. This will be our future work where we aim to refine the local criteria in a way to obtain optimal sequences, and by consequence to reduce the search time and the search space. Another extension to FAP will be the introduction of actions with conditional effects, where we thought the local criteria refinement would be harder to generate relevant sequences instead of generation a sequence for each possibility. The main search algorithm of FAP is the N-Best-Hill-Climbing which is complete and in which we can go back to revisit previous actions when needed. But our experiments have showed that the result is often achieved in the first pass.

## References

- B. Bonet, G. L., and Geffner, H. 1997. A robust and fast action selection mechanism for planning. *to appear in the AAAI-97 Proceedings*.
- Blum, A. L., and Furst, M. L. 1995. Fast planning through planning graph analysis. *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI95)* 1636–1642.
- Bonet, B., and Geffner, H. 2000. HSP: Heuristic search planner. *Entry at AIPS-98 Planning Competition, AI Magazine* 21(2).
- Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129:5–33.
- Gu  r  , E., and Alami, R. 2001. One action is enough to plan. *IJCAI* 17:439–444.
- Hoffman, J. 2001. FF: The fast-forward planning system. *AI Magazine* 22:57 – 62.
- Knoblock, C. 1994. Automatically generation abstractions for planning. *Artificial intelligence* 68(2):243–302.
- McDermott, D. 1996. A heuristic estimator for means ends analysis in planning. *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems. AIPS*.
- Pais, J., and Pinto-Ferreira, C. 1999. he n-best heuristic search algorithm. *In proceedings of the 18th Workshop of the UK Planning and Scheduling Special Interest Group PLANSIG99, England*.
- Sacerdoti, E. D. 1975. The nonlinear nature of plans. *In Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI-75)* 206–214.
- Tate, A. 1977. Interacting goals and their use. *In proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77)* 888–893.

# Marvin: Macro Actions from Reduced Versions of the Instance

Andrew Coles and Amanda Smith

Department of Computer and Information Sciences,  
University of Strathclyde,  
Livingstone Tower,  
26 Richmond Street,  
Glasgow,  
G1 1XH

email: `firstname.lastname@cis.strath.ac.uk`

## Abstract

Marvin is a forward-chaining heuristic-search planner. The basic search strategy used is similar to FF's enforced hill-climbing with helpful actions (Hoffmann & Nebel 2001); Marvin extends this strategy, adding extra features to the search and preprocessing steps to infer information from the domain.

## Introduction to Marvin

Marvin is a forward-chaining domain-independent planner that uses a relaxed-plan heuristic to guide its search. The name Marvin stands for Macro-Actions from Reduced Versions of the INstance and gives some insight into the way in which the planner works: it attempts to create a reduced instance of the problem with which it is presented, solve this smaller instance, and then use the solution to assist with solving the original problem.

## Basic Search Strategy

The basic search used is similar to FF's enforced hill-climbing with helpful actions (Hoffmann & Nebel 2001); Marvin extends this strategy, adding extra features to the search and preprocessing steps to infer information from the domain. This section details the modifications made to the search strategy.

When plateaux are encountered Marvin resorts to best-first search as opposed to breadth-first search—in practise this improves its performance but may increase the makespan of the plan.

To reduce the overheads incurred by memoising already-visited states no record is kept of visited states if search is progressing normally; however, should a plateau be encountered, the differences between states on the plateau and the state at the start of the plateau are memoised, and states whose difference has already been memoised are pruned.

To prune action choices Marvin constructs groups of symmetric objects (objects with identical properties), extracts one exemplar from each group and then prunes actions which involve any entities which are not the exemplar for their group; for example, in the gripper domain, if two balls are symmetrical in a given state it will only consider applying the pickup action to one of them.

Marvin can exploit the potential for concurrency in solution plans by considering, at each choice point, all of the actions that could be applied at the current time point ( $t$ ) before considering the actions that could be applied at the next time point (for non-temporal domains this is simply  $t + 1$ ). This approach increases the branching factor and could thus become very expensive during periods of exhaustive search; hence, during such periods the concurrency reasoning is suspended until the plateau is escaped. The steps to escape a plateau are then post-processed to reintroduce concurrency where possible.

## Instance Reduction

Before attempting to solve the problem instance with which it is presented, Marvin creates a smaller instance of the problem. This approach was motivated by the observation that small instances can be solved quickly and their solutions often contain action sequences similar to those in solutions for larger problem instances. Any knowledge that can be obtained inexpensively by solving a smaller instance will be valuable in solving the larger instance that was given to the planner.

Smaller instances are created using symmetry and almost-symmetry. Two objects are symmetric if, and only if, they share the same predicates in the initial and goal states: this is the definition of symmetry used previously by STAN version 3 (Fox & Long 1999). In many domains this reduction does not discard sufficient entities to create a significantly smaller problem, hence further pruning is desirable; this is achieved through the use of almost-symmetry. In this context two objects are almost symmetric if, and only if, the predicates defining them in the initial and goal state are of the same type and they differ only in groundings of one or more arguments of a the predicates. For example, in the problem below (where all predicates involving `package1` and `package2` are shown):

### Initial State

at package1 loc1  
at package2 loc2  
...

### Goal State

at package1 loc3

at package2 loc4

...

the two packages are ‘almost-symmetric’: they only differ by one binding in the initial state (the location they are at) and one in the goal state (their destination).

Using this definition of almost-symmetry the symmetry in the solution plan for these two entities will be captured, as well as strict symmetry in the problem: if two objects share the same predicates in the initial state (even if the groundings of these predicates differ) it is likely that the same, or a similar, plan can be used to achieve the required goals for both objects.

When the extraction of groups of related objects is completed a new smaller problem instance is created by taking one exemplar from each related group and including only the predicates whose entities are wholly contained within this set of exemplars; the smaller instance is then solved, using the search algorithm described in the previous section, to generate a solution plan.

The plan generated to solve the smaller instance is processed to produce macro-actions. Partial-order lifting is used to extract independent threads of execution in the plan; after extraction independent threads are made into individual macro-actions and are added to the list of actions to be used in planning to solve the original instance. Whilst adding actions does increase the branching factor the additional actions often assist in the planning process as they encapsulate a previously-successful strategy for solving a similar problem.

It should be noted that for some domains—for example, freecell—the reduced problem is unsolvable; in such situations it is usually the case that the problem is proven unsolvable very quickly: the goals do not appear in the relaxed planning graph. For situations in which the goals are present in the relaxed planning graph it is necessary to introduce an upper bound on the plan length allowed to ensure that an unreasonable amount of time is not spent solving the smaller instance; in practise this does not prevent Marvin from generating useful macro-actions as preliminary experiments show large macro-actions are often too specialised to a certain task and are therefore not reusable.

## Plateau-Escaping Macro-Actions

Solutions to planning problems often contain a given sequence of actions more than once; if finding this reused action sequence corresponds to exhaustive search a lot of unnecessary search effort is expended in repeatedly attempting to find this action sequence. Marvin attempts to improve on the plateau behaviour of previous forward-chaining planners by memoising the action sequence which successfully lead from the start of a plateau to a strictly-better state; these memoised action sequences form what are known as plateau-escaping macro-actions. To reduce the overheads of having a greater number of actions to consider at each state these plateau-escaping macro-actions are only considered when plateaux are encountered: in normal search only the original actions from the domain, and any actions derived from the solution to the reduced instance, are used.

When solving the reduced instance any plateau-escaping macro-actions devised are stored for use when later solving the original problem; this has the useful side-effect of discovering efficacious escape macros with less computational effort—it is less computationally expensive to perform the plateau-escaping search on the reduced instance of the problem. Furthermore, since the reduced instance is derived from the original problem instance, it is often the case that the heuristic breaks down when solving the reduced instance in some of the places it breaks down when solving the original problem instance.

As with the macro-actions created from the reduced version of the instance the plateau-escaping macro-actions have a partial order lifted out, the aim of which is to improve the concurrency within them, reducing the makespan. Once this processing has taken place the segment of plan which escaped the plateau is replaced with the macro-action: the macro-action may exploit concurrency which the original plan segment did not.

## Transformational Operators

Transformation operators are those operators that transform a certain property of an object but leave other objects unchanged; for example, the action move in the driverlog domain:

**pre:**

at (truck loc1)  
linked(loc1 loc2)

**add:**

at (truck loc2)

**del:**

at (truck loc1)

transforms the ‘at’ property of trucks. The reusability of macro-actions is adversely affected by transformation operators, as they often appear in chains of varying lengths; consequently, abstraction of the length of these chains is required if the macro-action is to be as reusable as possible.

Generating sequences of transformational operators is a shortest path problem, which can be solved by a specialist solver. Marvin currently recognises transformational operators by looking for a common fingerprint; however, in the future TIM (Long & Fox 2000) will be used to provide a method through which these operators can be identified in a more-robust manner.

When transformational operators have been identified an all-pairs shortest-path reachability analysis is done, during which the best route between two states is stored; then, static predicates for all pairwise reachable states are added to the initial state so that Marvin can plan as if the states were all linked. When an action is later selected for application the main algorithm simply asks the sub-solver for the action sequence required to achieve the desired effect.

## ADL

Marvin supports ADL natively; that is, without creating distinct STRIPS actions for each of the possible ADL action

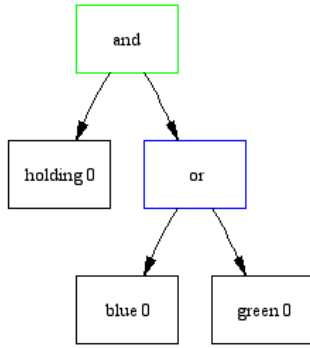


Figure 1: Example Satisfaction Tree

groundings. ADL support was written for the purpose of solving the competition ADL domains—without it, due to the nature of the STRIPS compilations provided, Marvin would not have been able to construct any reusable macro actions.

ADL preconditions are dealt with through the logical reduction of each operator’s preconditions to form a ‘Satisfaction Tree’. The idea is to create a tree where the leaves are predicates (or negations of predicates) and the internal nodes are either conjunction or disjunction nodes (AND or OR); then, predicates either help a given ground action become applicable (if they appear as positive predicate leaves in its satisfaction tree) or hinder its applicability (if they appear as negative predicate leaves). The tree is formed by recursively applying the following rules to each action’s preconditions:

$$\begin{aligned}
(\forall x f(x)) &\Rightarrow (f(x_0) \wedge \dots \wedge f(x_n)) \\
(\exists x f(x)) &\Rightarrow (f(x_0) \vee \dots \vee f(x_n)) \\
(a \Rightarrow b) &\Rightarrow (\neg a \vee b) \\
(\neg(T_0 \wedge \dots \wedge T_n)) &\Rightarrow (\neg T_0 \vee \dots \vee \neg T_n) \\
(\neg(T_0 \vee \dots \vee T_n)) &\Rightarrow (\neg T_0 \wedge \dots \wedge \neg T_n)
\end{aligned}$$

The first two of these simply compile out the existential quantifiers dynamically; the third is a logical reformation of the implies operator; the final two, forms of De Morgan’s duality law, are used to force any negation into the subexpressions, and eventually to the predicates.

Figure 1 shows an example satisfaction tree for an action in an imaginary domain in which objects can only have a certain action applied to them if they are being held and are either blue or green.

ADL effects are handled in a similar manner to preconditions, in that they form ‘Effect Trees’; there are differences, however, due to the differing semantic structure between Preconditions and Effects: Effect Trees do not contain OR nodes; instead they introduce ‘When’ nodes. When nodes have two child branches - a condition branch (which is, itself, a satisfaction tree) and an effect branch (which is an effect tree). When an action is grounded any unconditional effects and effects contingent only on static predicates are associated with the ground action instance; sub-actions are then created to encapsulate any effects contingent on dynamic information.

The relaxed planning graph in Marvin is modified to account for the negative preconditions required by ADL. Before the ADL support was implemented a spike (Long & Fox 1999) for positive predicates was used; to build a relaxed planning graph forward from a given state the spike was initialised to contain the predicates in a given state and then grew as applied relaxed actions added predicates to it. To support negative preconditions a second spike was created; this spike is initialised to be empty and then any predicate present in the initial fact layer which is then, later, deleted is added to it. A negative precondition is then satisfied at a given layer in the relaxed planning graph either if it isn’t present in the initial fact layer or it has since appeared in the negative fact spike.

## Future Work

In the future Marvin will be extended to use the generic-type recognition knowledge provided by TIM (Long & Fox 2000). This will, amongst other things, improve its support for transformational operators by providing a flexible framework for their identification; also, it raises the possibility of using generic-type-derived heuristics to improve the discrimination between states when the relaxed plan graph heuristic reaches a plateau.

Marvin will also be extended to deal with Temporal Planning: as it already uses macro-actions and concurrency, much of the framework is already complete.

## References

- Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *IJCAI*, 956–961.
- Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.
- Long, D., and Fox, M. 1999. Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research* 10:87–115.
- Long, D., and Fox, M. 2000. Automatic synthesis and use of generic types in planning. In *Artificial Intelligence Planning Systems*, 196–205.

# A Petri net based representation for planning problems

Marcos Castilho  
André Guedes  
Tiago Lima  
João Marynowski  
Razer Montaña

Departamento de Informática,  
Federal University of Paraná,  
Curitiba, Brazil

Luis Künzle  
Fabiano Silva  
CPGEL,  
CEFET-PR,  
Curitiba, Brazil

## Introduction

In this paper we propose a Petri net based representation for planning problems. The motivation for this is that Petri nets are a formal tool useful to model and analyse domains involving true parallelism, concurrency, conflicts, and causal relations which are beyond the scope of classical planning.

In (Silva, Castilho, & Künzle 2000) we presented a way to translate the plan graph into an acyclic Petri net. This would already serve as a basis for our desired analysis on non-classical planning. However, that translation kept the same redundancies of the plan graph. It just translate propositions and actions in the plan graph to places and transitions in the Petri net.

In this first translation we didn't explore the dynamics of Petri nets. In the approach proposed in this paper we show the construction of the Petri net directly from the description of the problem. In this new structure, we give another view about the mutex relation and maintenance actions. We give details about this in section .

In Petri nets, a planning problem corresponds to a submarking reachability problem. This is known to be EXPspace-hard (Lipton 1976; Esparza & Nielsen 1994) in the general case. Fortunately, our net is an acyclic one and in this case we are in the NP-complete case (Stewart 1995), which is what we expected. Anyway, to solve the reachability problem is not straightforward and due to lack of space we refer the reader to (Rauhamaa 1990). In this paper we focus on the structure of our model.

In the next section we recall the basis of Petri nets. Then we present the construction of a Petri net directly from the description of the planning problem. Finally we present some concluding remarks.

## Petri Nets, Reachability and the Petriplan algorithm

A Petri net (Murata 1989) is a 4-tuple  $N = (P, T, Pre, Post)$  where  $P = \{p_1, p_2, \dots, p_n\}$  is a finite set of places,  $T = \{t_1, t_2, \dots, t_m\}$  is a finite set of transitions,  $Pre : P \times T \rightarrow \mathbb{N}$  is the input incidence function and  $Post : P \times T \rightarrow \mathbb{N}$  is the output incidence

function. A Petri net with a given initial marking is denoted by  $(N, M_0)$  where  $M_0 : P \rightarrow \mathbb{N}$  is the initial marking.

The Petri net dynamics is given by *firing* enabled transitions, whose occurrence corresponds to a state change of the system modelled by the net. A transition  $t$  of a Petri net  $N$  is enabled for a marking  $M$  iff  $M \geq Pre(., t)$ . This enabling condition, expressed under the form of an inequality between two vectors, is equivalent to  $\forall p \in P, M(p) \geq Pre(p, t)$ .

Only enabled transitions can be fired. If  $M$  is a marking of  $N$  enabling a transition  $t$ , and  $M'$  the marking derived by the firing of  $t$  from  $M$ , then  $M' = M + Post(., t) - Pre(., t)$ . Note that the firing of a transition  $t$  from a marking  $M$  derives a marking  $M'$ :  $M \xrightarrow{t} M'$ .

We can generalise this formula to calculate a new marking after firing a sequence  $s$  of transitions. Let us consider a matrix  $C = Post - Pre$ , called Petri net *incidence matrix*, and a vector  $\bar{s}$ , called *characteristic vector* of a firing sequence  $s$  ( $\bar{s} : T \rightarrow \mathbb{N}$ , such that  $\bar{s}(t)$  is the number of times that transition  $t$  appears in the sequence  $s$ ). The number of transitions in  $T$  defines the dimension of the vector  $\bar{s}$ . Then, firing a sequence  $s$  of transitions from  $M$ , a new marking  $M_g$  is calculated by the *fundamental equation* of  $N$ :

$$M_g = M + C \cdot \bar{s}. \quad (1)$$

We can use the fundamental equation to determine a vector  $\bar{s}$  for a given net  $N$  and two markings  $M$  and  $M_g$ . The satisfying solution must be a nonnegative integer vector, and it is only a necessary condition for  $M_g$  to be reachable from  $M$ . This condition becomes necessary and sufficient for *acyclic Petri nets*, a subclass of Petri nets that have no directed circuits (Murata 1989).

The reachability relation between markings of a firing transition can be extended, by transitivity, to the reachability of the firings of a transition sequence. Thus, in a Petri net  $N$ , it is said that the marking  $M_g$  is reachable from the marking  $M$  iff there exists a sequence of transitions  $s$  such that:  $M \xrightarrow{s} M_g$ . The reachability set of a marked Petri net  $(N, M_0)$  is the set  $\mathcal{R}(N, M_0)$  such that  $(M \in \mathcal{R}(N, M_0)) \Leftrightarrow (\exists s M_0 \xrightarrow{s} M)$ .

We call the *reachability problem* for Petri nets the problem of determining if a given marking  $M_g$  is reachable from  $M_0$ . The *sub-marking reachability problem* for a given

sub-marking  $M_s$  consists of determining if exists a marking  $M_g$  that is reachable from  $M_0$  and  $M_s \subset M_g$ , where  $M_g \in \mathcal{R}(N, M_0)$ . In (Rauhamaa 1990) we have several different techniques to solve it.

The *Petriplan* algorithm consists in two steps: first, the construction of a Petri net from the description of the planning problem; then find a sequence of transitions firings that solves the reachability problem. In the next sections we explore the construction of the net directly from the description of the problem, taking profit of the representational power of a Petri net.

### The plan net

In this section we modify the structure of our Petri net defined in (Silva, Castilho, & Künzle 2000) and define what we call the *plan net*, which is simply a Petri net obtained directly from the description of the problem exploring the representational power of Petri nets. We need however to explain two important points before showing the construction technique.

First of all, let's consider the representation of propositions. In the beginning of the construction of the net a place represents a proposition. During the process when it is found that a proposition is a precondition of more than one action, we just copy the place. It may happen that a place will be copied several times.

Now let's consider the possible inconsistencies between actions. In the plan graph this means to look for the mutex relation between action in some layer. When this is found the actions are marked as mutex, i.e., these two actions cannot be executed at the same time. This forces the copy of the entire layer to a new one using maintenance actions. In a certain sense the conflict is not completely solved, just in the "search for a solution" phase the two actions are ordered.

In our case the proposal is to have no maintenance actions. What we do is to refine the mutex relation. We relate two actions in five different ways, not only two (mutex and not mutex). Let  $x$  and  $y$  be two actions. We define the following:

- $(x \parallel y)$ : they are totally independent, that is, they may happen even in parallel. This is the "not mutex" in the plan graph sense. It may be possible to have only  $x$ , only  $y$ ,  $x$  followed (or preceded) by  $y$  and  $x$  and  $y$  in parallel;
- $(x \neg y)$ :  $x$  has as effect the negation of some effect of  $y$ . This way  $x$  and  $y$  may occur in any order, but not in parallel;
- $(x \not\prec y)$ :  $x$  has as effect the negation of some precondition of  $y$ . So  $x$  could not occur before or in parallel with  $y$ ;
- $(x \not\prec y)$ :  $y$  has as effect the negation of some precondition of  $x$ . So  $y$  could not occur before or in parallel with  $x$ ;
- $(x \diamond y)$ :  $x \not\prec y$  and  $x \not\prec y$ . The given actions may occur just each one alone or with a third action between them.

This is an important difference between the graph and the plan net. The price for this is that we need to find out the correct kind of relation between two actions. The algorithm is based on a graph structure called *graph of static inconsistencies*, which is a graph whose nodes are actions and there

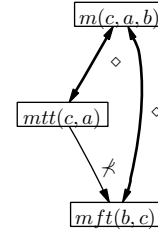


Figure 1: Graph of static inconsistencies for the first layer.

is an edge of type  $t$  linking  $x$  and  $y$  if  $x$  is related with  $y$  with respect with relation  $t$ . Observe that  $(x \diamond y)$  is the stronger case. The process of construction of this graph has the same computational cost of finding all the static mutex relations in the plan graph.

Now we are in condition to show the algorithm to construct the plan net. This process follows the idea of the construction of the plan graph. It begins with marked places representing the initial state.

We enter then in a loop looking for the places representing the final state. This loop has three phases, which are described in details below.

Phase 1: we add transitions representing all possible actions whose preconditions are already in the net. If some place is already a precondition of some other transition create a copy of this place. This copy is not needed only in the case whether the consequence of the action is the negation of that precondition been copied. This copy will be linked with the transition been added. This phase will define a layer, i.e., all possible actions that may be fired simultaneously.

Phase 2: we construct the graph of static inconsistencies for the transitions in the last generated layer (figure 1). It is constructed as we explained above. This graph will guide the construction of the *control structure of the net*. This is a Petri net containing all possible sequences of non inconsistent actions present in the last generated layer. The places here are not associated with propositions, they are just control places. We merge this structure in the net. The merge process is to include copies of the actions appearing in the control structure that are not in the original net. But we do not need to copy the places representing preconditions of the actions been copied. For example in figure 2 the action  $mft(b, c)^0$  was copied to  $mft(b, c)^1$ , but both share the same preconditions  $f(b)^1$ ,  $f(c)^0$  and  $ot(b)^0$ . At the end of this phase we have a Petri net containing all possible ways of executing the actions without any conflict in this layer. Figure 2 shows the resulting net.

We must say that the notion of layer in the Petri net is different from that in the plan graph. Here, a layer may contain actions happening in more than one instant of time, whereas in the plan graph each layer is associated with only one instant of time. Due to the process of construction based on the graph of static inconsistencies we can warrant that there is no static inconsistent sequences of actions in each branch of the net in this layer.

Phase 3: if the net contains places representing the goal state we enter phase 3, i.e., we will look for a solution. That

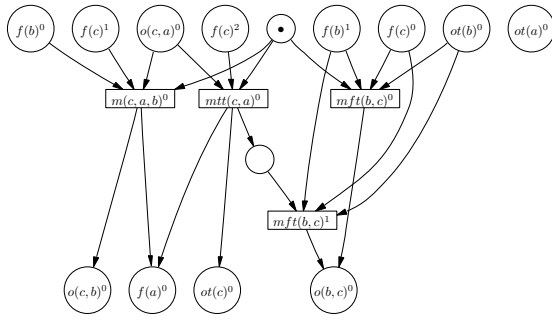


Figure 2: The first layer for Sussman anomaly with control structure.

means to find a flow in the net which puts tokens in the places representing the goal state. This is the reachability problem in Petri nets. As said, we refer to (Rauhamaa 1990) for the complexity of this problem. If such a flow exists, then it is a (possibly parallel) plan. In the other case, we return to phase 1. In our example there is no such a flow. So we must return one more time to phase 1 and 2. For lack of space we will not show the figures. Now in phase 3 the flow exists. Figure 3 the final Petri net for the Sussman anomaly. This net is a simplified version containing just the paths which reach some goal state place.

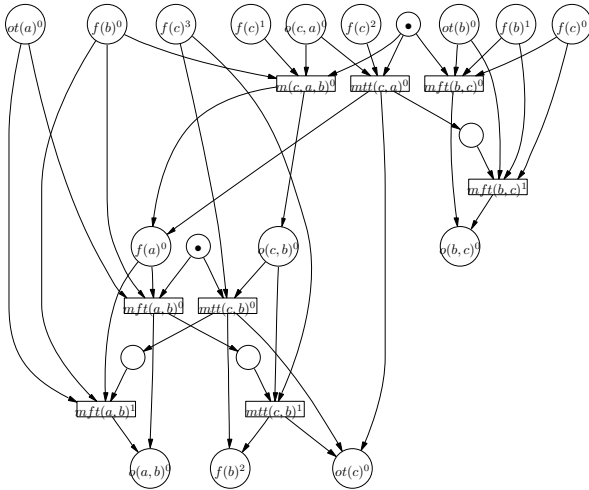


Figure 3: Final Petri net for the Sussman anomaly.

## Discussion

Relations between Petri nets and planning problems were former investigated by (Murata & Nelson 1991) and (Mieller & Fabiani 2000). The first use a general cyclic predicate-transition Petri net. The problem is that the necessary and sufficient condition of equation 1 is broken, and the only way to solve the reachability problem is to use the reachability graph, which leads to an enumerative search for a solution.

The second approach defines a cyclic coloured Petri net,

in which each place corresponds to a logical predicate describing actions preconditions or effects. The operators instantiation is made by token colours. The theoretical model obtained for the resulting planning problem is in fact more compact than ours, but it presents the same problem of exhaustive search, as in (Murata & Nelson 1991).

In our approach, however, we have a simpler acyclic place-transition Petri net, with necessary and sufficient conditions to use the equation 1 to find a solution to the planning problem. This paper modifies our first presentation of the *Petriplan* algorithm (Silva, Castilho, & Künzle 2000) by taking profit of the dynamics of the Petri net thus reducing the structure.

Finally, the method proposed in this paper permits to construct a Petri net representation of the planning problem. As others methods, we can find a solution to the planning problem, in our case using reachability algorithms. The classical way is to start an exhaustive search, just as *Graphplan* does. However, as we have an acyclic Petri net, the matrix representation of the fundamental equation can be viewed as a constraint satisfaction problem, which can be solved using several methods, as integer programming, SAT, among others.

## References

- Esparza, J., and Nielsen, M. 1994. Decidability issues for Petri nets - a survey. *Bulletin of the European Association for Theoretical Computer Science* 52:245–262.
- Lipton, R. J. 1976. The reachability problem requires exponential space. Technical report, Dept of Computer Science, Yale University. research report 62.
- Mieller, Y., and Fabiani, P. 2000. Planning with Petri nets. In *Proc. of RJCIA-2000*.
- Murata, T., and Nelson, P. 1991. A predicate-transition net model for multiple agent planning. *Information Sciences* 57-58:361–384.
- Murata, T. 1989. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77(4):541–580.
- Rauhamaa, M. 1990. A comparative study of methods for efficient reachability analysis. Technical Report A 14, Digital Systems Laboratory, Helsinki University of Technology. <http://citeseer.nj.nec.com/245545.html>.
- Silva, F.; Castilho, M.; and Künzle, L. 2000. Petriplan: a new algorithm for plan generation (preliminary report). In *Proc. of IBERAMIA/SBIA-2000*, 86–95. Springer-Verlag.
- Stewart, I. A. 1995. Reachability in some classes of acyclic Petri nets. *Fundamenta Informaticae* 23(1).

# SGPlan: Subgoal Partitioning and Resolution in Planning\*

Yixin Chen, Chih-Wei Hsu, and Benjamin W. Wah

Department of Electrical and Computer Engineering  
and the Coordinated Science Laboratory  
University of Illinois, Urbana-Champaign  
Urbana, IL 61801, USA  
{chen, chsu, wah}@manip.crhc.uiuc.edu

## Abstract

We have developed SGPlan, a planner that competes in the Fourth International Planning Competition. SGPlan partitions a large planning problem into subproblems, each with its own subgoal, and resolves inconsistent solutions of subgoals using our extended saddle-point condition. Subgoal partitioning is effective because each partitioned subproblem involves a substantially smaller search space than that of the original problem. We have developed methods for the detection of reasonable orders among subgoals, an intermediate goal-agenda analysis to hierarchically decompose each subproblem, a search-space-reduction algorithm to eliminate irrelevant actions in subproblems, and a strategy to call the best planner to solve each bottom-level subproblem. Currently, SGPlan supports PDDL2.1 and derived predicates, and algorithms for supporting time initiated facts and ADL are under development.

## OVERALL ARCHITECTURE

By formulating a subproblem in such a way that each has one goal state, SGPlan partitions a planning problem into subproblems, orders the subproblems according to a sequential resolution of its subgoals, and finds a feasible plan for each goal fact. Using the extended saddle-point condition and constrained search, new constraints are enforced to ensure that facts and assignments in a later subgoal are consistent with those of earlier subgoals.

Figure 1 shows the architecture of our planner. In the global level, we select a suitable order for the planner to solve the partitioned subgoals, introduce artificial global constraints to enforce that the solution of one subgoal solved later does not invalidate that of an earlier subgoal, and resolve violated global constraints using the theory of extended saddle points. In the local level, we perform a hierarchical decomposition of first-level

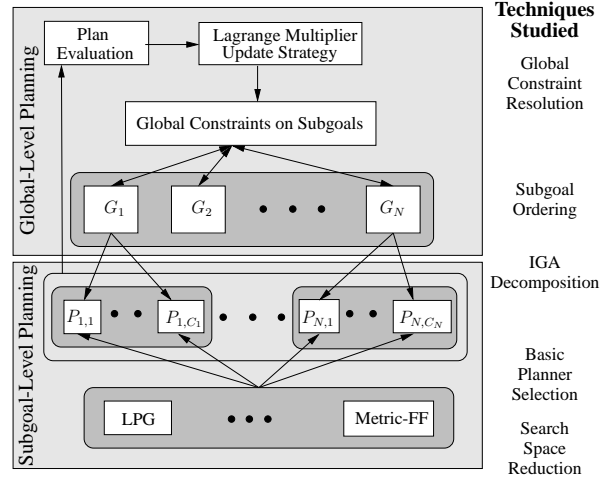


Figure 1: The architecture of SGPlan.

subgoals, prune irrelevant facts and actions before calling a basic planner, and choose a suitable basic planner for solving the second-level subproblem.

Figure 2 presents the pseudo code of our planner. Based on the subgoals identified, we partition the problem into  $N$  subproblems  $G_1, \dots, G_N$ , one for each subgoal, and order the subproblems appropriately. For  $G_i$ , we perform an intermediate-goal-agenda (IGA) analysis to decompose it into  $C_i$  smaller subproblems  $P_{i,1}, \dots, P_{i,C_i}$ . For each second-level subproblem, we perform subspace-reduction analysis to reduce its search space and choose a suitable planner (called *basic planner*) to solve it. Finally, we evaluate the composed plan and update the Lagrange multipliers.

Our approach is different from incremental planning (Koehler & Hoffmann 2000) that uses a goal agenda. In incremental planning, a planner maintains a set of target facts, adds goal states incrementally into the target set, and extends the solution by using the new target set. This means that a goal state will always be satisfied once it is satisfied. However, it may be more expensive to solve subsequent problems, since the search space increases as more goal states are added. Moreover, it is difficult to tell which goals should be satisfied before others. In contrast, SGPlan always involves only

\*Research supported by the National Aeronautics and Space Administration Grant NCC 2-1230 the National Science Foundation Grant ITR 03-12084.  
Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.



```

1. procedure SGPlan
2.   compute the partial orders among subgoals;
3.   generate an initial ordered list of subgoals;
4.   set  $iter \leftarrow 0$ ;
5.   repeat
6.     for each goal fact in the subgoal list
7.       find the intermediate goal facts;
8.       generate an IGA agenda;
9.       for each entry in the IGA agenda
10.        call search space reduction procedure and
11.         eliminate irrelevant actions;
12.        call basic planner to solve the subproblem;
13.     end_for
14.     end_for
15.     if (plan  $z$  found is feasible)
16.       evaluate the solution plan;
17.       decrease some Lagrange multipliers;
18.     else increase Lagrange multipliers  $\gamma$  on unsatisfied
19.       global constraints;
20.      $iter \leftarrow iter + 1$ ;
21.     if ( $iter \% \tau == 0$ ) dynamically re-order the subgoals;
22.   until no change on  $z$  and  $\gamma$  in an iteration;
23. end_procedure

```

Figure 2: The pseudo code of SGPlan.

one goal fact in a subproblem. Therefore, the search space of the subproblems is not increasing, and irrelevant actions in each subproblem can be pruned.

## GLOBAL-LEVEL PLANNING

### Subgoal Ordering and Global Constraints

When dependent subgoals are evaluated sequentially, it is possible that a subgoal evaluated later may invalidate the results of a subgoal evaluated earlier, and the earlier subgoal has to be re-evaluated. Although such conflicts may be unavoidable, appropriately ordered subgoals can significantly reduce the occurrences of such conflicts. Intuitively, difficult subgoals should be resolved before easier ones.

It is non-trivial to find an optimal order that minimizes the conflicts among subgoals. In fact, it may be more computationally expensive to find the best order than solving the problem itself. In SGPlan, we have developed three heuristics for partial ordering of subgoals that can be computed efficiently (Step 2 of SGPlan).

The first level is called *reasonable ordering* proposed in (Koehler & Hoffmann 2000). Suppose goal fact  $A$  is ordered before  $B$  in the subgoal list, but after we get a plan that achieves  $A$ , we cannot achieve  $B$  without invalidating  $A$  first. Then the search for achieving  $A$  first is wasted, and it is more efficient to achieve  $B$  before  $A$ . We use an algorithm in FF2.2 (Koehler & Hoffmann 2000) to find such reasonable orders.

For goal pairs not ordered by reasonable ordering, we apply a second level of ordering called *irrelevance ordering*. Based on backward relevance analysis (discussed in the next section), we compute the number of irrelevant actions of each goal fact, and order  $A$  before  $B$  if  $A$  has less irrelevant actions. The idea is to resolve

more difficult subgoals, with less irrelevant actions.

For goal pairs not ordered by the first two levels, we apply the third level of ordering called *precondition ordering*. Specifically, for  $A$  and  $B$  with the same number of irrelevant actions that cannot be ordered by reasonable ordering, we order  $A$  before  $B$  if  $n_p(A) > n_p(B)$ . Here,  $n_p(A)$  is the minimum number of preconditions of those supporting actions:

$$n_p(A) = \min_{a \in S(A)} n_{pre}(a), \quad (1)$$

where  $S(A)$  is the set of all actions that support goal fact  $A$ , and  $n_{pre}$  is the number of preconditions of action  $a$ . Again, the idea is that more difficult goals, with larger  $n_p$ , should be resolved first.

For pairs of subgoals that are not involved in any of the three levels or ordering, we randomly order them. At the beginning of a search, we randomly generate a total ordering of the goal facts that satisfy the three levels of partial orders (Step 3) and periodically generate new total orders during the search (Step 19).

To identify conflicts among solutions of subgoals, we define a global constraint so that the solution plan of a subgoal will not invalidate the goal fact of another subgoal. Each global constraint in SGPlan is a binary constraint that indicates whether conflicts exist or not.

### Resolution of Global Constraints

The planning problems studied in SGPlan are defined in mixed space with nonlinear objective and constraints that may be procedural and not in closed form. SGPlan implements a search to find extended saddle points in the Lagrangian space of a problem (Chen & Wah 2003; Wah & Chen 2003). The extended saddle-point condition (ESPC) states that solution points in mixed space that are local optima of the objective and that satisfy all the constraints must satisfy ESPC. The condition is defined on a Lagrangian function that consists of the sum of the objective and the constraints weighted by Lagrange multipliers, where an extended saddle point is a point that is a local minimum of Lagrangian function with respect to the original variable space and a local maximum of the function with respect to the Lagrange-multiplier space.

An important property of ESPC is that the condition is true for all Lagrange multipliers larger than a minimum threshold. Hence, finding points that satisfy ESPC can be implemented iteratively, with an inner loop that looks for local minimum of the Lagrangian function, and an outer loop that looks for any Lagrange multipliers larger than the critical threshold. The property also allows a search looking for extended saddle points to be partitioned into multiple searches, each looking for a local extended saddle point for a partitioned problem (Steps 6-12 of Figure 2), and an outer loop that resolves the global constraints across the subproblems (Step 17).

A direct implementation of ESPC in a search algorithm may get stuck in an infeasible region when the objective is too small or when the Lagrange multipliers

and/or constraint violations are too large. To address this issue, SGPlan performs periodic decreases of Lagrange multipliers in the Lagrangian space in the outer loop, in addition to ascents (Step 16).

## SUBGOAL-LEVEL PLANNING

### Subgoal-Level Decomposition

Sometimes the subproblems after first-level partitioning by subgoals are still too large to be solved quickly. An obvious approach to reduce this complexity is to further partition the subproblem into smaller ones.

Given subgoal  $G$  after first-level partitioning, we propose to identify some “hidden” intermediate second-level subgoals (or facts) that must be true in any plan that achieves  $G$  from a given initial state (Steps 7 and 8). These facts allow us to construct an intermediate goal agenda (IGA), which is an ordered list of agenda entries, each containing a set of intermediate facts.

From a fixed initial state  $S$ , we define the following relationship between two facts  $A$  and  $B$ .  $A$  is an intermediate goal before  $B$ , denoted as  $A \preceq_{IGA} B$ , if the planning graph starting from  $S$  cannot achieve  $B$  without achieving  $A$  first. We construct the planning graph similar to that in Graphplan, with the following two changes: a) we do not compute any mutual exclusion relations; b) we forbid the insertion of  $A$  into the planning graph at any level (thereby also forbidding the insertion of any actions having  $A$  as a precondition). If  $B$  is not in the planning graph after the construction of the graph, then we have  $A \preceq_{IGA} B$ .

Based on the intermediate facts, we detect the  $\preceq_{IGA}$  orders among them and construct a directed graph showing their partial orders. We then identify an agenda of sets of facts that must be true in any plan of  $G$ .

SGPlan determines dynamically whether partitioning should be further carried out, depending on whether a subgoal  $G$  is easy enough to be resolved quickly using the IGA agenda. If subgoal  $G$  is to be partitioned, SGPlan further uses symmetry-group detection to see if a path can be constructed from the current facts to the subgoal:  $f_0 \rightarrow f_1 \rightarrow \dots \rightarrow G$ , where  $f_0, f_1, \dots$  are all in the same symmetry group as that of  $G$ . It then partitions the problem of achieving  $G$  from  $f_0$  into  $N$  subproblems:  $f_0 \rightarrow f_1, f_1 \rightarrow f_2, \dots, f_{N-1} \rightarrow G$ .

Our approach is different from existing approaches for finding intermediate facts (Koehler & Hoffmann 2000) that expand a search space from the goal state and find some indispensable pre-conditioning facts. Since the initial state is not specified, there is no way to tell to what depth the backward expansion should stop. In contrast, our method considers both the initial and the goal states in determining whether an intermediate fact is critical and always stops in finite levels of expansions. In addition, we detect the partial orders among these facts and form an agenda to avoid unachievable intermediate states, which could occur in previous methods.

### Search-Space Reduction

After partitioning a subproblem into easier second-level subproblems, we can often eliminate many irrelevant actions in their search space before solving them. Such a reduction is generally not applicable to planning problems that are not partitioned because in most cases all actions in their search space are relevant.

We have designed a polynomial-time *backward relevance analysis* to exclude some irrelevant actions before applying any planner to solve a subproblem (Step 10). Given a subproblem to be solved, we maintain an *open list* of unsupported facts, a *close list* of relevant facts, and a *relevance list* of relevant actions. In the beginning, the open list contains only the subgoal facts of the subproblem, and the relevance list is empty. In each iteration, for each fact in the open list, we find all the actions supporting that fact and not already in the relevance list. We then add these actions to the relevance list, and add the action preconditions that are not in the close list to the open list. We move a fact from the open list to the close list when it is processed. The analysis ends when the open list is empty. At that point, the relevance list will contain all possible relevant actions, while excluding those irrelevant actions.

Since partitioned subproblems usually have similar structures, we learn suitable rules for subproblem solving during a search. After a number of trial-and-error, SGPlan records some suitable heuristics and parameters that lead to the successful resolution of subgoals and use them in solving other subproblems.

### Basic-Planner Selection

Our current implementation of SGPlan uses a modified Metric-FF planner for basic planning and only invokes LPG when the modified planner fails. We have developed new algorithms and modified heuristic functions in the enhanced Metric-FF to fully support derived predicates, temporal planning, and time initiated facts (still under development).

## References

- Chen, Y. X., and Wah, B. W. 2003. Automated planning and scheduling using calculus of variations in discrete space. In *Proc. Int'l Conf. on Automated Planning and Scheduling*, 2–11.
- Koehler, J., and Hoffmann, J. 2000. On reasonable and forced goal ordering and their use in an agenda-driven planning algorithm. *J. of AI Research* 12:339–386.
- Wah, B. W., and Chen, Y. X. 2003. Partitioning of temporal planning problems in mixed space using the theory of extended saddle points. In *Proc. IEEE Int'l Conf. on Tools with Artificial Intelligence*, 266–273.

# Planning in PDDL2.2 Domains with LPG-TD

Alfonso Gerevini Alessandro Saetti Ivan Serina Paolo Toninelli

Dipartimento di Elettronica per l'Automazione

Università degli Studi di Brescia

Via Branze 38, 25123 Brescia, Italy

{gerevini,saetti,serina}@ing.unibs.it

## Introduction

LPG-TD is an extension of the LPG planner (Gerevini, Saetti, & Serina 2003; 2004) that can handle most of the features of PDDL2.2 (Edelkamp & Hoffmann 2003), the standard planning language of the 4th International Planning Competition (IPC-4).<sup>1</sup> In particular, LPG-TD is an incremental fully-automated planner generating plans for problems in domains involving:

- STRIPS actions;
- durative actions;
- actions and goals involving numerical expressions;
- operators with universally quantified effects;
- operators with existentially quantified preconditions;
- operators with disjunctive preconditions;
- operators with implicative preconditions;
- timed initial literals (deterministic unconditional exogenous events);
- predicates derived by domain axioms;
- maximization or minimization of complex plan metrics.

Like the previous version of LPG, the new version is based on a stochastic local search in the space of particular “action graphs” derived from the planning problem specification. In LPG-TD, this graph representation has been extended to deal with the new features of PDDL2.2, as well to improve the management of durative actions and of numerical expressions (already supported by PDDL2.1 (Fox & Long 2003)).

In the following, we briefly describe the main novelties of LPG-TD, which include some new techniques for planning problems involving *timed initial literals* and *derived predicates*, and some general improvements of all phases of the planner (pre-processing, search and post-processing).

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

<sup>1</sup>The “TD” extension in the name of the planner is an abbreviation of “Timed initial literals and Derived predicates”, the two main new features of PDDL2.2.

## Handling Timed Initial Literals

Timed initial literals represent facts (predicates instantiated with constants) that become true or false at certain time points, independently of the actions in the plan. They correspond to particular exogenous events known by the planner (Edelkamp & Hoffmann 2003). A fact can become true or false several times through different timed initial literals, defining a set of disjoint *temporal windows* where the fact holds. For example, the first problem of the *Satellite* domain in IPC-4 has two timed initial literals

```
(at 139.00 (visible antenna0 satellite0)),  
(at 219.04 (not (visible antenna0 satellite0)))
```

defining a single temporal window for the fact

```
(visible antenna0 satellite0).
```

According to PDDL2.2, the fact involved by a timed initial literal can appear in the preconditions of an action, while it can never appear in its effects. We call such preconditions *timed preconditions*, and we represent them as particular nodes of the action graph. If a plan action  $a$  has a timed precondition  $p$  of type “overall” involving a fact  $f$ ,  $p$  is satisfied when the interval identified by the start time and the end time of  $a$  is contained into at least one temporal window associated with  $f$ . Similar conditions can be defined for the other possible types of preconditions in a durative action.

Essentially, an unsatisfied timed precondition involving a fact  $f$  in  $a$  is treated by either (i) removing  $a$  from the plan under construction, or making some changes to the plan that make the execution of  $a$  compatible with a temporal window associated with  $f$ , i.e., by (ii) appropriately postponing the start time of  $a$ , or (iii) removing one or more actions that permit to decrease the start time of  $a$ .

In the new version of LPG, the graph-based plan representation, the pre-processing phase (reachability analysis and computation of the “mutex relations”), and the search techniques have been extended to perform such plan modifications when dealing with unsatisfied timed preconditions.

## Handling Derived Predicates

Derived predicates are predicates that can not be achieved directly by the domain actions. A derived predicate  $P(\bar{x})$  is true at a certain time  $t$  during the execution of a plan iff it

can be derived from the facts that are true at time  $t$  through a set of rules specified in the domain formalization. Each of these rules is of the form

*if  $\phi(\bar{x})$  then  $P(\bar{x})$ ,*

where  $\bar{x}$  is a tuple of variables, and  $\phi(\bar{x})$  a logical formula (a precise syntactic and semantic definition of domain rule is given in (Edelkamp & Hoffmann 2003)).

A typical example of derived predicate in the Blocksworld domain is *above*, which can be derived by using the following rule:

*if  $(on(x, y) \vee \exists z \text{ above}(x, z) \wedge \text{above}(z, y))$   
then  $(\text{above}(x, y))$ .*

In PDDL2.2, a derived predicate can be a precondition of an action or a goal of the planning problem, which we call *derived precondition* (we treat problem goals as preconditions of a special final action). A derived precondition of an action  $a$  is satisfied if it is implied by the domain rules and the facts that are true when  $a$  is executed.

Essentially, an unsatisfied derived precondition  $d$  in  $a$  is treated by either (i) removing  $a$  from the current plan, or (ii) adding one or more actions that modify the set of the facts that are true when the action can be executed in the plan, so that  $d$  becomes true by applying of one or more domain rules. For example, consider a simple Blocksworld problem where the initial state is

$(on\text{-}table\ a), (on\text{-}table\ b), (on\ c\ b)$

and the goal is  $(above\ a\ b)$ . When the domain rule of the previous example is available, it is easy to see that the goal can be achieved by just adding to the (initially empty) plan the action  $stack(a, c)$  making  $(on\ a\ c)$  true.

In the new version of LPG, the graph-based plan representation, the pre-processing phase (reachability analysis and computation of the mutex relations), and the search techniques have been extended to take possible domain rules into account.

### Further Extensions

In addition to the treatment of timed initial literals and derived predicated, the new version of our planner includes several revisions and extensions with respect to the version that took part in the previous competition. Such changes concern the pre-processing phase, the search phase, and post-processing phase of the planner. In the following, we give a list of them.

#### Pre-processing

- The algorithm for computing mutex relations has been revised to make it faster than the original algorithm described in (Gerevini, Saetti, & Serina 2003).
- Some actions are automatically identified as “useless actions”, and they can be pruned away at parsing time or they can be neglected during search.
- The computation of the reachability information for numerical domains has been improved to derive more accu-

rate information that are exploited by the heuristic function evaluating the search neighborhood.

#### Search

- We have developed new heuristics for evaluating the search neighborhood specialized for the different variants of a planning domain supported by PDDL2.2.
- The basic local search strategy (Walkplan) has been extended with a “tabu list” helping to escape from local minima.

#### Post-processing

- We have developed a technique for increasing the degree of parallelism in the plans generated by LPG for domains with durative actions and numerical expressions. This is done by an algorithm that, starting from the set of the actions forming the plan and their ordering constraints identified by the planner, tries to reduce the plan makespan.

Finally, at the time of writing, the development of LPG-*TD* is still in progress. In particular, we are experimenting a pre-processing technique for the automatic ordering of the problem goals, and we are developing an extension of the representation for handling actions involving conditional effects.

### Acknowledgments

We would like to thank all previous members of the LPG team and particularly Alberto Bettini, Marco Lazzaroni, Sergio Spinoni.

### References

- Edelkamp, S, Hoffmann, J. 2003. PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition Technical Report N. 194, Albert Ludwigs Universität Institut für Informatik, Freiburg, Germany.
- Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR* 20:61–124.
- Gerevini, A., Saetti, A., and Serina, I. Planning through Stochastic Local Search and Temporal Action Graphs in LPG. 2003. *JAIR* 20:239–290.
- Gerevini, A., Saetti, A., and Serina, I. 2004. An Empirical Analysis of Some Heuristic Features for Local Search in LPG. In *Proceedings of ICAPS-04*.

# The Workings of CRIKEY - a Temporal Metric Planner

**Keith Halsey**

University of Strathclyde  
Glasgow, UK  
keith.halsey@cis.strath.ac.uk

## Abstract

Described here is the temporal metric planner CRIKEY as it competed in the International Planning Competition 2004. CRIKEY separates out the planning and scheduling parts of temporal planning problems, and detects where these two sub-problems are too tightly coupled to be separated completely. In these cases it solves the sub-problems together. The domains of the competition are looked at to see where these interactions occur.

## Introduction

CRIKEY is a forward heuristic search planner based closely on MetricFF (Hoffmann 2002) and implemented in Java1.4. In a similar fashion to MIPS (Edlekamp & Helmert 2000), it separates the planning and scheduling where it can, however it solves the two problems together where such a relaxation will fail. It is this combining of the problems only where necessary and the reasoning associated with it that distinguishes it from other similar planners (and where the focus of the research lies). It can detect these cases in the domain and act accordingly. I am only interested in where the interaction and separation of sub-problems will prevent a solution being found, and not where this separation leads to an inferior quality of solution. CRIKEY is complete and sound but not optimal (either in time or the specified metric). It will however make an attempt to minimise the number of actions in a plan.

## Capabilities

CRIKEY was written to work with the PDDL2.1 (Fox & Long 2001) models of metrics and time. It can deal with both temporal aspects (i.e. durative actions) and metrics resources. More formally, it can parse and plan with PDDL domains with the `:typing`, `:fluents`, and `:durative-actions` requirements. Unfortunately, currently it can not make use of any of the ADL constructs or the new language features (namely, timed initial literals or derived predicates).

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

## Architecture

The architecture of CRIKEY is shown in Figure 1. It first looks at the domain for where planning and scheduling could potentially interact. Then it performs forward heuristic search using a relaxed plan graph. The mini-scheduler makes sure that a schedulable plan is passed into the scheduling phase. This consists of lifting a partial order plan from the totally ordered plan, and then turning this into a temporal plan. Crucially, there is no feedback from the scheduling phase to the planning phase, therefore the planner must produce a plan that the scheduler can schedule.

## Technical Details

### Planning

CRIKEY finds a plan through forward heuristic search similar to FF (Hoffmann & Nebel 2001). During planning, temporal information is ignored. The search strategy is enforced hill climbing, that is, once a better state is found, search proceeds from that state without backtracking. Best first search is used on plateaus, where all neighbouring states are no improvement on the current state. If enforced hill climbing fails, best first search is attempted from the initial state. This is complete and so theoretically should find a plan.

The heuristic value is the length (number of actions) of a relaxed plan where the delete effects are ignored. The relaxed plan is from the current state to the goal state and is easily extracted from a relaxed planning graph.

As in FF, only helpful actions are considered in the enforced hill climbing. Helpful actions are actions which appear in the first layer of the relaxed planning graph and are also in the relaxed plan.

### Scheduling

A greedy algorithm (Moreno *et al.* 2002) works backwards through the totally ordered plan finding causal links between the starts and ends of actions to form a partially ordered plan. Links are either  $\leq$  or  $<$  (in which case a minimum value equal to the tolerance value must separate the two end points). These are put into an STN upon which Floyd-Warshalls Algorithm is to calculate the actual time of the actions in the partially ordered plan.

The algorithm must not only look for orderings based on logical conditions, but also for orderings due to metric con-

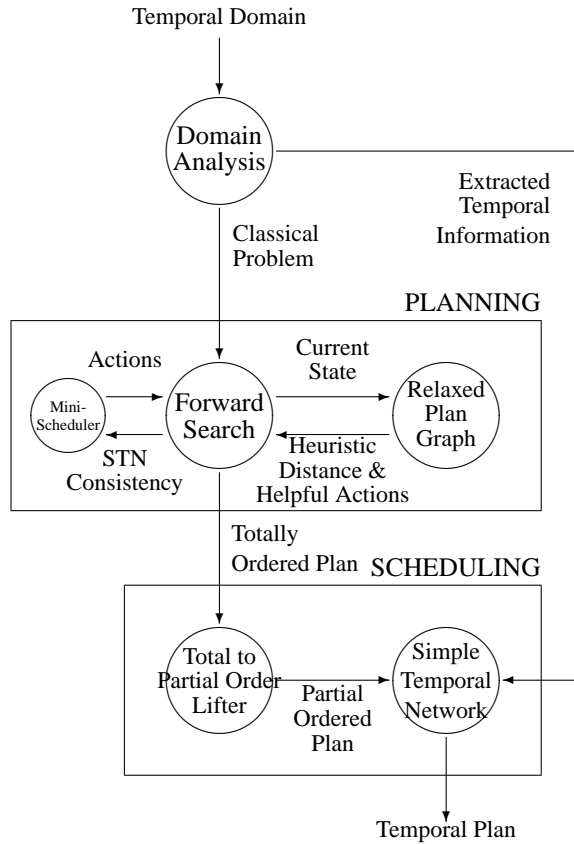


Figure 1: Architecture Overview of CRIKEY

straints. For a  $>$  or  $\geq$  resource constraint, just enough producers of that resource are ordered before it, assuming that all consumers that precede it in the totally ordered plan, occur before it in the partially ordered plan. The same is true for  $<$  or  $\leq$  conditions, apart from the roles of consumer and producers are reversed. Whilst this is conservative, it must be sound as the totally order plan is correct (at worst, the partial order will be the same as the total order).

The next section details how it is impossible to produce an unschedulable plan.

## Interactions

In cases where the planning and scheduling interact, precautions must be made to ensure that a plan is not produced which is unschedulable. This can happen where the actions *must* happen in parallel (as opposed to the more common case where actions *can* happen in parallel if they do not interfere). That is to say, one or more actions (called “content actions”) must happen whilst another (the “envelope action”) is executing. If there is not enough time to execute the contents during the envelope, then an unschedulable plan is produced.

These cases are detected in advance by looking for “potential envelopes” - actions which allow other actions only

to happen during their duration. These happen where:

$$\begin{aligned}
 & (end_{cond} \setminus start_{add} \neq \emptyset \wedge start_{add} \setminus end_{cond} \neq \emptyset) \\
 & \vee start_{del} \cap end_{cond} \neq \emptyset \\
 & \vee add_{start} \cap del_{end} \neq \emptyset
 \end{aligned}$$

We shall name three states,  $s_1$ , the state immediately before the start of the action,  $s_2$ , the state immediately after the start, and  $s_3$  the state immediately after the end of the action. An action applicable in  $s_2$  and not in  $s_1$  must have been achieved by the at start add effects (since there are no negative conditions, it could not have been achieved by an at start delete effect). Taking it further, there are no actions that could be applied in  $s_2$  and not in  $s_3$  which could not have been applied in  $s_1$ , apart from those achieved by the at start add effects and then deleted by the at end delete effects. Alternatively, an action could be achieved by the start effect, and the effects of this action needed to achieve the end conditions. They are called potential envelopes since (at the moment) there is no effort to find out if there are any content actions that must go in these envelopes.

As stated, where there are potential envelopes, there is the potential to produce an unschedulable plan. To avoid this, envelope action are split into two separate actions, a start action containing the start conditions and effects, and an end action containing the end conditions and effects. Invariants become conditions of the end action, and, if not achieved by the start effects, also of the start action. An end action cannot be applied until its corresponding start action is in the plan, and a plan is not valid until all the start actions in the plan also have their corresponding end actions in the plan.

On putting a start action into the plan, a mini-scheduler is associated with this action. This mini-scheduler consists of a Simple Temporal Network, a set of content actions (initially empty) and a set of orderings between these actions. The mini-schedulers use the same algorithms as the main scheduling part of CRIKEY. Any (content) actions which are now considered, must be checked against this mini-scheduler to ensure that if they must go in the envelope, the STN is consistent (that is to say that there is enough time to execute the action). If not, then the action is not considered applicable, and that branch is removed from the search space. When the envelope’s end action is chosen, the mini-scheduler is then discarded. Figure 2 is pseudo-code for the mini-scheduler. As can be seen, invariants are protected whilst an envelope’s start has been chosen but not its end action. No other action may delete these invariants until that action has completed.

## Competition Domains

Unfortunately, none of the domains in the 2004 competition in their purest form (that is, without the new features compiled out) contained any envelopes (i.e. no actions *had* to happen in parallel) and so in all problems the planning and scheduling were relatively loosely coupled. This means that CRIKEY could not show off its mini-scheduling capabilities to cope with these situations. It is hoped that after the competition, the other competing planners will become available and it will be possible to compare them with CRIKEY on domains which do contain such situations.

1. Check  $A_{cond}$  are satisfied. If not, return false.
2. Check  $A_{del}$  do not delete invariants in the list of invariants. If not, return false.
3. If  $A$  is a start of an envelope
  - (a) Create a new mini-scheduler for  $A$  and add to list of mini-schedulers.
  - (b) Add  $A$ 's invariants to the list of invariants.
4. Else If  $A$  is an end of an envelope
  - (a) Remove  $A$ 's mini-scheduler from the list of mini-scheduler.
  - (b) Remove  $A$ 's invariants from the list of invariants.
5. For Each envelope  $E$  currently open
  - (a) Get orderings for  $A$  in  $E$ .
  - (b) If no orderings, return true.
  - (c) Add orderings to the STN.
  - (d) Return the consistency of the STN.

Figure 2: Algorithm to decide whether an action  $A$  is applicable

Envelopes were present in versions of the domains where timewindows and deadlines had been compiled down from PDDL2.2 to PDDL2.1. These envelopes are present in the newly created dummy actions to enforce the constraints and lasted the length of the plan. As the envelope lasts the length of the plan, the mini-scheduler for each dummy action is active throughout the planning process. This is highly inefficient and not what the mini-schedulers are designed to solve. However, it still makes sure that an unschedulable plan is not passed to the scheduler.

Since there were no domains particular to CRIKEY's designed purpose and strengths, not much development of CRIKEY was performed whilst the competition was running, except to correct bugs in the code and parser. It is thought that not being able to handle ADL was not such a disadvantage as CRIKEY would probably have only performed an equivalent compilation internally.

## References

- Edlekamp, S., and Helmert, M. 2000. On the implementation of mips. In *Proceedings from the 4th Artificial Intelligence Planning and Scheduling (AIPS), Workshop on Decision-Theoretic Planning*, 18–25. Breckenridge, Colorado: AAAI-Press.
- Fox, M., and Long, D. 2001. PDDL2.1: An extension to PDDL for expressing temporal planning domains. Technical report, University of Durham, UK.
- Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.
- Hoffmann, J. 2002. Extending FF to numerical state variables. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02)*, 571–575.
- Moreno, D.; Oddi, A.; Borrajo, D.; Cesta, A.; and Meziat, D. 2002. Integrating hybrid reasoners for planning and

scheduling. In *Proceedings from the 21st UK Planning and Scheduling Special Interest Group (PlanSIG'02)*, 179–189.

# TP4 '04 and HSP<sub>a</sub>\*

**P@trik Haslum**  
Linköping University  
pahas@ida.liu.se

## Abstract

TP4 and HSP<sub>a</sub>\* are optimal temporal planners, though they assume a semantics for temporal planning problems that differs somewhat from the PDDL2.1 standard. Both use regression, and automatically extracted admissible heuristics to inform search: their only difference is that HSP<sub>a</sub>\* invests more time in computing a more accurate heuristic. Two new tricks were added to the planners to cope with some domains in the 2004 planning competition. The more interesting of those is a two-stage optimization scheme which speeds up planning in domains with highly uneven action durations.

## Introduction

The TP4 and HSP<sub>a</sub>\* planners find temporal plans for STRIPS problems with durative actions. The plans found are optimal *w.r.t.* makespan, *i.e.* the total execution time of the plan, and the planners are also able to ensure that the plan does not violate certain kinds of resource constraints.

TP4 participated in the 2002 planning competition, where it may be said to have ended up second-to-last (although it rightfully deserved the last place)<sup>1</sup>. The version of TP4 participating in the 2004 competition is a reimplementaion of essentially the same planner. The new implementation, having been designed to be a flexible experimental platform for variations of the basic planning algorithm (such as HSP<sub>a</sub>\*) rather than an efficient implementation of a single algorithm, is somewhat slower than the earlier version.

This paper focuses on two points: First, the semantics that TP4/HSP<sub>a</sub>\* assume for planning problems (which differs from the PDDL2.1 standard) and second, new tricks that were added to the planners to address problems encountered in the competition domains.

## The Semantics of Planning Problem Specifications

Put somewhat pointedly, TP4 does not accept PDDL2.1 input<sup>2</sup>. For practical purposes it uses the same syntax, but durative actions and fluents are interpreted in a manner that differs from the PDDL2.1 specification (Fox & Long 2003).

<sup>1</sup>This is my interpretation: Such a strict ordering of planners was not an official result of the competition.

<sup>2</sup>The same applies to HSP<sub>a</sub>\*.

## Durative Actions

The semantics that TP4 assumes for durative actions are essentially those introduced by Smith and Weld (1999) for their TGP planner.

An action  $a$  has preconditions  $pre(a)$ , positive (added) and negative (deleted) effects  $add(a)$  and  $del(a)$ , which are all sets of atoms, and a duration  $dur(a) > 0$ . Preconditions that are not deleted by the action are termed persistent preconditions, *i.e.*  $per(a) = pre(a) - del(a)$ . For action  $a$  to be executable over a time interval  $[t, t + dur(a)]$ , atoms in  $pre(a)$  must be true at  $t$ , and atoms in  $per(a)$  must remain true (*i.e.* not interfered with) over the entire interval. Effects of the action take place at some point in the interior of the interval, and thus can be relied on to hold at the end point. This respects the “no moving target” rule of PDDL2.1, but in a different way: instead of requiring plans to explicitly separate an action depending on a condition from the effect that establishes the condition, TP4’s semantics requires that *change takes place in a time interval*.

TP4’s interpretation makes durative actions strictly less expressive than in PDDL2.1, where effects can be specified to take place exactly at the start or end of an action. In particular, it does not support actions that make a condition true only during their execution (*i.e.* add the atom at the start of the action and delete it again at the end), which prevented TP4 from solving any of the problems with timed initial literals, since the compilation of those makes use of this type of effect.

## Resources

TP4 does not deal with fluents but with resources, specifically resources of two kinds: A *reusable* resource is one that actions “borrow” some quantity of during their execution, but the total amount of the resource (free and in use), does not change over time. A *consumable* resource is one that each action may either consume or produce some quantity of, thus changing the total (and free) amount of the resource over time<sup>3</sup>.

Resources of both kinds can be modelled in PDDL2.1 using fluents and certain “patterns” of action conditions and

<sup>3</sup>This is similar to what is called a *reservoir* by Laborie (2001). A reservoir, however, can be both borrowed and consumed/produced.



effects, and TP4 identifies resources in a problem by looking for these patterns. For example, an action with the effects (at start (decrease  $f$   $m$ )) and (at end (increase  $f$   $m$ )), and the condition (over all ( $\geq f$  0)), uses the fluent  $f$  as a reusable resource<sup>4</sup>. However, in PDDL2.1 it is possible to express the same resource restriction also in other ways, *e.g.* by having actions that use the resource increase  $f$  at start, decrease it at end and require that  $f \leq F$ , for some static fluent  $F$  representing the capacity of the resource. TP4's resource finding procedure had to be extended with several new patterns to correctly identify resources in the `umts` competition domain.

TP4 requires consumable resources to be decreasing, *i.e.* actions may only consume (not produce) them<sup>5</sup>. It also does not allow a resource to be used both as a reusable and a consumable. Among the competition domains involving resources, only the `settlers` domain failed to meet these restrictions.

### TP4/HSP<sub>a</sub>\* Planning Algorithm

TP4 searches for plans using temporal regression, *i.e.* backchaining from the problem goals over actions that are positioned in time so that they form a schedule, not just a sequence. The search is done using IDA\*, including standard enhancements such as cycle checking and a bounded transposition table, and guided by an admissible heuristic, which is derived from the problem specification. The planner is described in more detail in (Haslum & Geffner 2001).

HSP<sub>a</sub>\* is very similar: the only difference is that it invests more time in computing a more accurate heuristic before the search. It does so by solving a relaxed version of the problem and recording information discovered in the search. TP4 computes the  $H^2$  heuristic (which assigns an estimated cost to all possible sets of at most 2 subgoals, see Haslum and Geffner (2001) for the definition of  $H^m$ , for  $m = 1, \dots$ ). HSP<sub>a</sub>\* does likewise, but improves on this by computing part of the  $H^3$  heuristic (assigning a better estimated cost to some sets of 3 or fewer subgoals) by searching the AND/OR graph corresponding to the definition of the  $H^3$ . The details are described in a forthcoming paper<sup>6</sup>.

In the competition domains, TP4 and HSP<sub>a</sub>\* showed little difference in performance, with two exceptions: in the `umts` domain, HSP<sub>a</sub>\* did a little better than TP4, while in the `airport` domain, it was much worse.

### New Tricks

Apart from the already mentioned extension to the resource finding procedure, TP4 learned two new tricks during the competition<sup>7</sup>:

<sup>4</sup>TP4 also allows actions to use atoms as unary reusable resources, identified by a similar pattern.

<sup>5</sup>If both consumption and production of the same resource are allowed, and actions may test if a resource is depleted (without changing it), the planning problem becomes undecidable (Helmert 2002). Whether this is the case also when such “resource tests” are disallowed is not completely clear.

<sup>6</sup>Submitted to ECAI.

<sup>7</sup>Again, the same applies to HSP<sub>a</sub>\*.

### Irrelevance Detection

Detection (by standard reverse unreachability) and removal of irrelevant atoms and actions helped speed up the planner on some problems in the `airport` domain, but was used for all domains since the time overhead for this analysis is quite small.

### Two-Stage Optimization

When using IDA\* with temporal regression, the cost bound tends to increase by the gcd (greatest common divisor) of action durations in each iteration, except for the first few iterations<sup>8</sup>. In the `satellite` domain, durations differ by large amounts and are also specified with a high resolution (*e.g.* one action may have a duration of 7.89 and another a duration of 122.03) which means the gcd is very small (on the order of  $\frac{1}{100}$ ). Combined with the fact that the difference between the initial heuristic estimate of the solution cost (makespan) of a problem and the actual optimal cost is in this domain often large, this results in an almost astronomical number of IDA\* iterations being necessary to find the optimal solution.

To counter this problem, the following “two-stage optimization” scheme was introduced:

1. First, all action durations are rounded up to the nearest integer.
2. Then, the resulting problem is solved using the standard TP4 method. The cost (makespan) of the solution is an upper bound on the optimal solution cost of the original problem.
3. Finally, action durations are restored to their original values, and a branch-and-bound search, starting from the known upper bound, is used to find the optimal solution.

The solution found in step 2 is always a valid solution to the original unmodified problem<sup>9</sup>. The solution cost (makespan), however, may be greater than the optimal solution cost for the unmodified problem. Thus it is an upper bound. The branch-and-bound search in step 3 is carried out on the unmodified problem (with the original, fractional, action durations), so the final solution found in this search is the optimal solution to the original problem. Thus, two-stage optimization does not compromise the optimality of the planner overall.

Rounding action durations up to integer values increases their gcd to at least 1 (a substantial improvement from  $\frac{1}{100}$ ),

<sup>8</sup>TP4 treats action durations as rationals: by the gcd of two rationals  $a$  and  $b$  is meant the greatest rational  $c$  such that  $a = mc$  and  $b = nc$  for integers  $m$  and  $n$ . Note that the planner does *not* compute the gcd of action durations and use this to increment the cost bound. The bound is in each iteration increased to the cost of the least costly node that was not expanded due to having a cost above the bound in the previous iteration (*i.e.* standard IDA\*). That this frequently happens to be (on the order of) the gcd of action durations is an (undesirable) effect of the branching rule used to generate the search space.

<sup>9</sup>This fact is due to the semantics that TP4 ascribes to durative actions. It does not hold for arbitrary problems interpreted according to the PDDL2.1 semantics.

so the search in step 2 is much faster than what an IDA\* search on the unmodified problem would be. Since the branch-and-bound search does not suffer from the problem of small gcd's and the upper bound obtained from step 2 tends to be quite close to the optimal cost, step 3 is relatively quick, and the total time less than that taken by plain TP4.

In principle, there seems to be no reason why in step 1 action durations could not be rounded up to produce a gcd greater than 1, even going as far as assigning unit duration to all actions (essentially turning the problem into a non-temporal problem). Whether this would make the two-stage optimization scheme more effective is a topic that may be investigated in the future.

Among the competition domains, two-stage optimization was effective only in (temporal variants of) the *satellite* domain, and it was not used for any other domain.

## References

- Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of AI Research* 20:61 – 124. <http://www.cs.washington.edu/research/jair/contents/v20.html>.
- Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *Proc. 6th European Conference on Planning (ECP'01)*, 121 – 132.
- Helmert, M. 2002. Decidability and undecidability results for planning with numerical state variables. In *Proc. 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02)*, 303 – 312.
- Laborie, P. 2001. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. In *Proc. 6th European Conference on Planning (ECP'01)*, 205 – 216.
- Smith, D., and Weld, D. 1999. Temporal planning with mutual exclusion reasoning. In *Proc. 16th International Joint Conference on Artificial Intelligence*, 326 – 333.

# Fast Downward

## Making use of causal dependencies in the problem representation

Malte Helmert and Silvia Richter

Institut für Informatik, Albert-Ludwigs-Universität Freiburg  
Georges-Köhler-Allee, Gebäude 052, 79110 Freiburg, Germany  
{helmert, srichter}@informatik.uni-freiburg.de

### Abstract

Fast Downward is a propositional planning system based on heuristic search. Compared to other heuristic planners such as FF or HSP, it has two distinguishing features: First, it is tailored towards planning tasks with *non-binary* (but finite domain) state variables. Second, it exploits the *causal dependency* between state variables to solve relaxed planning problems in a hierarchical fashion.

Fast Downward is a planning system based on heuristic state space search, in the spirit of HSP or FF (Bonet & Geffner 2001; Hoffmann & Nebel 2001). It makes use of the *causal graph* (or CG) heuristic, introduced in an ICAPS 2004 paper (Helmert 2004). In this extended abstract, we aim at providing a high-level overview of Fast Downward, emphasizing the features that are not described in the CG article. While the CG heuristic was introduced for pure STRIPS domains, Fast Downward is capable of dealing with the complete propositional, non-temporal part of PDDL. In other words, it handles arbitrary ADL constructs and derived predicates (axioms).

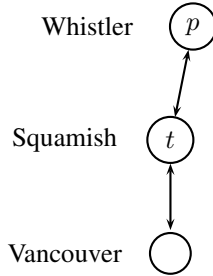


Figure 1: A simple planning task. Get the ICAPS participant  $p$  to Vancouver, using the taxi  $t$ .

The key feature of the CG heuristic — and the origin of Fast Downward’s name — is the use of *hierarchical decomposition* to solve relaxed planning tasks. To illustrate this, consider the planning task in Fig. 1: The objective is to move the ICAPS participant  $p$  from Whistler ( $W$ ) to Vancouver ( $V$ ), using a taxi ( $t$ ) initially located at Squamish ( $S$ ).

The CG heuristic solves this problem hierarchically. The high-level goal is to change the state of the participant from

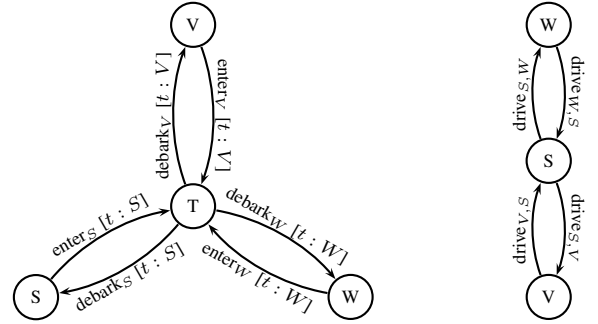


Figure 2: Domain transition graphs for the participant  $p$  (left) and taxi  $t$  (right).

“at Whistler” to “at Vancouver”. The easiest way to do this is to board the taxi at Whistler and debark at Vancouver; at this point we do not care that these actions are not immediately applicable. This plan is found by looking at the ICAPS participant’s *domain transition graph*, a directed graph depicting the ways in which  $p$  can change locations (Fig. 2). The different locations or *states* of  $p$  form the nodes of the graph, while the arcs correspond to operators affecting these states, annotated with their preconditions.

To estimate the cost of the “high-level plan”  $p : W \rightsquigarrow T \rightsquigarrow V$ , the heuristic solver inserts steps to satisfy the preconditions of the two operators by recursive invocations of the same algorithm. The transition  $p : W \rightsquigarrow T$  requires the taxi to be at Whistler, as evidenced by the labeling of that arc in  $p$ ’s domain transition graph. So we recursively find a (one-step) plan to move the taxi from its initial location Squamish to Whistler. Because there are no conditions on the transitions of the taxi (Fig. 2), there is no further recursion. We have thus computed that the cost of changing the state of the participant from  $W$  to  $T$  is 2, counting one action for the transition itself and one for the recursively calculated set-up cost. Similarly, we compute that the second transition  $p : T \rightsquigarrow V$  is 3, because the taxi is now located in Whistler and thus needs two actions to get to Vancouver, in addition to the one action required to move  $p$  out of the taxi. Adding the transition costs together, the CG heuristic approximates the goal distance as  $5 = 2 + 3$ .

Observe that state transitions of the passenger are condi-

tioned on the state of the taxi, while the converse is not the case. We say that state variable  $p$  is *causally dependent* on state variable  $t$ . The set of causal dependencies of a planning task defines the *causal graph* of that task. Hierarchical decomposition is most suited to planning domains with acyclic causal graphs. In fact, the CG heuristic can only be calculated for tasks with acyclic causal graphs, and hence Fast Downward's heuristic estimator breaks causal cycles for the purposes of the heuristic estimator, by ignoring (some) operator preconditions. Contrast this relaxation to HSP's approach of ignoring (some) operator effects.

We hope that this small example provides the reader with some intuition of the basic ideas of the CG heuristic. Again, we point to the reference for a detailed exposition (Helmert 2004). In the following, we discuss the overall structure of the Fast Downward planner, emphasizing aspects that go beyond the STRIPS planner described in the conference paper.

## Structure of the planner

Fast Downward currently consists of three independent programs:

1. the translator (written in Python),
2. the preprocessor (written in C++), and
3. the search engine (also written in C++).

To solve a planning task, the three programs are called in sequence; they communicate via text files. We have found that this clear separation facilitates simultaneous development of the planner by several people in its current prototype stage. Of course the current state of affairs leads to some inefficiencies, especially when solving easy or moderately difficult planning tasks. For hard tasks, runtime is typically dominated by the search engine.

## Translator

The *translator* has the following responsibilities:

- Compiling away (most) ADL features.
- Grounding the operators and axioms.
- Converting the propositional (binary) representation to one with multi-valued state variables.

It is commonly known that some features of ADL can be compiled away easily, i.e. without significantly increasing the problem representation, while others cannot (Nebel 1999). However, in the presence of axioms, all ADL constructs except for conditional effects can be translated to STRIPS quite easily.

Fast Downward applies the following transformations, in order, to simplify the problem representation:

- Translate implications to disjunctions and translate all conditions to negation normal form (NNF).
- Compile away universal quantifiers in conditions.
- Translate conditions to prenex normal form.
- Translate the quantifier-free part of conditions into disjunctive normal form.

- Split operators or axioms with disjunctive conditions into several operators or axioms, and split conditional effects with disjunctive conditions into several effects.

All these transformations are fairly basic, except maybe for the elimination of universal quantifiers explained now. Using the equivalence  $\forall x\varphi \equiv \neg\exists x\neg\varphi$ , the translator introduces a new axiom for  $\exists x\neg\varphi$  and replaces the universally quantified condition  $\forall x\varphi$  by the literal  $\neg\text{new-axiom}(\overline{V})$ , where  $\overline{V}$  is the set of free variables in  $\exists x\neg\varphi$ .

For example, the blocked axiom in the Promela domain contains the condition (ignoring types):

$$\forall t(\forall s'\neg\text{trans}(q, t, s, s') \vee \text{blocked-trans}(p, t)).$$

This is translated to the condition  $\neg\text{new-axiom}(p, q, s)$ , where  $\text{new-axiom}(p, q, s)$  is defined as:

$$\exists t(\forall s'\neg\text{trans}(q, t, s, s') \vee \text{blocked-trans}(p, t)),$$

which is translated to NNF, resulting in:

$$\exists t(\exists s'\text{trans}(q, t, s, s') \wedge \neg\text{blocked-trans}(p, t)).$$

After all transformations, all conditions are essentially simple conjunctions of literals (the remaining existential quantifiers can be considered action, axiom or effect parameters), so the resulting planning task is expressed in STRIPS with negation plus universal conditional effects and axioms.

For such planning tasks, efficient grounding is comparatively easy. Following the idea of Mips (Edelkamp & Helmert 1999), we avoid instantiating operators which can never be applied by first computing the set of propositions which are reachable in a *relaxed exploration*, ignoring negative conditions and effects. This amounts to the evaluation of a set of Horn logic rules derived from the actions and axioms. For example, the above axiom corresponds to the rule

$$\text{new-axiom}(p, q, s) :- \text{trans}(q, t, s, s').$$

The final translation step consists of replacing the set of binary state variables obtained by grounding with a smaller set of finite domain state variables capturing the same information. This is done by synthesizing invariants of the planning task, again using the algorithm of Mips.

To illustrate this, the variables  $p$  and  $t$  of our earlier example task are derived from the original PDDL representation by use of invariants. Specifically, the invariant

$$\exists_{=1} l : \text{taxi-at}(l),$$

justifies replacing the three binary variables  $\text{taxi-at}(V)$ ,  $\text{taxi-at}(S)$  and  $\text{taxi-at}(W)$  by the variable  $t$  with domain  $\{V, S, W\}$ .

## Preprocessor

The *preprocessor* is responsible for:

- Computing the causal graph of the planning task.
- Computing the domain transition graphs for each state variable.
- Computing the *successor generator*, a data structure that supports efficiently computing the successor states of a world state. (We do not discuss the successor generator in detail.)

Computing the causal graph is straight-forward: Variable  $A$  depends on variable  $B$  iff there is an operator (axiom) with  $A$  as an effect (consequence) and  $B$  as a condition or other effect. One notable optimization is employed at this point: All variables which are not mentioned in the goal and on which the goal does not depend directly or indirectly can be eliminated. For example, in the PSR domain, all instances of the `upstream` axiom for which the first parameter is not a circuit breaker may be safely removed.

As noted before, an acyclic causal graph is required for the CG heuristic. Therefore, for the purposes of the domain transition graphs, we compute an *acyclic skeleton* of the causal graph, i.e. a maximal acyclic subgraph. Cycles are broken by removing the weakest edges; this means that every dependency is weighted according to how often it occurs in the operators, and the edges with least weight are removed iteratively, until no cycle remains.

The central part of the preprocessor is the computation of the domain transition graphs. The domain transition graph of a variable contains arcs for all operators or axioms affecting this variable. For example, the graph for  $p$  in Fig. 2 contains an arc from  $V$  to  $T$  because there exists an operator with precondition  $p = V$  and effect  $p = T$ , corresponding to the action of boarding the taxi in Vancouver. The arc is annotated with the condition  $t = V$  because the operator requires the taxi to be in Vancouver as an additional precondition. We would omit this condition if the causal link between  $p$  and  $t$  were not part of the acyclic skeleton of the causal graph computed earlier. Thus, this is the part of the planner where some preconditions get ignored.

The reference (Helmert 2004) explains the details of domain transition graph construction for basic STRIPS-like operators; we note that the conditional effects present in the more general case do not lead to complications because domain transition graphs deal with operators one effect at a time, and for unary operators effect conditions can safely be considered part of the operator precondition.

## Search Engine

After so much preprocessing, the actual search algorithm is not very mysterious. Fast Downward uses greedy best-first search, always expanding the node with the best heuristic estimate. The heuristic is computed from the domain transition graphs as follows: The goal distance of a state is taken to be the sum of the costs for all necessary changes of variables. The cost for changing the value of one variable  $V$  from  $v$  to  $v'$  is the sum of the costs for all transitions of  $V$  on the shortest path from  $v$  to  $v'$  in  $V$ 's domain transition graph, computed using Dijkstra's algorithm.

The cost for traversing a single arc in the domain transition graph — the arc weight in Dijkstra's algorithm — is one plus the *set-up cost* of the transition, the sum of the (recursively computed) costs for achieving all necessary preconditions according to the arc label.<sup>1</sup> This follows the informal description of the CG heuristic in the introduction.

<sup>1</sup>If the arc corresponds to the derivation rule of an axiom, not to an action, then the weight is just the set-up cost, without adding 1.

## Helpful Actions

As a further enhancement, Fast Downward incorporates the CG counterpart of FF's helpful actions: The planner collects all operators that correspond to domain transition graph arcs which contribute to the heuristic estimate of the given state. It then checks which of these operators are applicable in the current state. These form the set of helpful actions in that state. This set can be empty although the heuristic estimate is finite, because domain transition graphs do not respect all operator preconditions, as discussed before.

The overall best first search algorithm integrates helpful actions by maintaining two separate open lists; all states are first inserted into the first open list. When a state from this list is expanded, the "helpful" successors are generated and the state is inserted into the second open list. When a state from the second list is expanded, its "non-helpful" successors are expanded. The search control always selects that open list for expansion which has generated fewer search states so far. This means that if an average state encountered during search has 4 helpful and 40 other successors, the first open list is selected ten times out of eleven, thus biasing the exploration towards helpful actions.

## Fast Diagonally Downward

As a final twist, we have also implemented a modified version of the search engine which combines CG heuristic and FF heuristic. This is based on the observation that CG and FF heuristic perform badly in different planning domains (Helmert 2004). Combining the forward and downward thrust by a simple vector addition, we have called this variant of the Fast Downward planner *Fast Diagonally Downward*.

Fast Diagonally Downward's search engine computes both the CG and FF heuristic for each state, as well as making use of helpful actions of both kinds. It uses separate open lists for the two heuristics, alternately expanding the node preferred by the FF estimate and the node preferred by the CG estimate. Newly generated states are always added to both open lists, making the approach different to simply running two planners in parallel. The hope is that the heuristics can lead each other out of their respective local minima, and indeed in some domains the combined approach works better than either of the original heuristics.

## References

- Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.
- Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proc. ECP-99*, 135–147.
- Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proc. ICAPS 2004*.
- Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.
- Nebel, B. 1999. What is the expressive power of disjunctive preconditions? In *Proc. ECP-99*, 294–307.

# SATPLAN04: Planning as Satisfiability

Henry Kautz

Department of Computer Science & Engineering  
University Of Washington  
Seattle, WA 98195 USA

SATPLAN04 is a updated version of the planning as satisfiability approach originally proposed in (Kautz & Selman 1992; 1996) using hand-generated translations, and implemented for PDDL input in the blackbox system (Kautz & Selman 1999). Like blackbox, SATPLAN04 accepts the STRIPS subset of PDDL and finds solutions with minimal parallel length: that is, many (non-interfering) actions may occur in parallel at each time step, and the total number of time steps in guaranteed to be as small as possible.

Also like blackbox, SATPLAN works by:

1. Constructing a graphplan-style (Blum & Furst 1995) style planning graph up to some length  $k$ ;
2. Translating the constraints implied by the graph into a set of clauses, where each specific instance of an action or fact at a point in time is a proposition;
3. Using a general SAT solver to try to find a satisfying truth assignment for the formula;
4. If the result is unsat or time out, increment  $k$  and repeat;
5. Otherwise, translate the solution to the SAT problem to a solution to the original planning problem;
6. Postprocess the solution to remove (some of the) unnecessary actions.

The final step is useful because the SAT translation of the planning graph does not guarantee that every action proposition that is true in the solution is actually needed in order to achieve the goals of the original plan.

SATPLAN04 supports four different encoding styles, “action-based”, “graphplan-based”, “skinny action-based”, and “skinny graphplan-based”, based on the classes of clauses included in the encoding. Classes of clauses are:

1. An action implies its preconditions.
2. A fact implies the disjunction of the actions that have it as an effect (including “no op” actions) at the previous time slice.

3. An action implies each of the disjunctions of the actions at the previous time slice that add each of its preconditions.
4. Actions with conflicting preconditions and effects are mutually exclusive.
5. Actions for which mutual exclusion can be inferred using graphplan’s constraint propagation algorithm are mutually exclusive.

“Graphplan-based” encodings use classes (1) and (2), while “action-based” encodings use class (3). “Skinny” encodings include class (5) while non-skinny encodings include both (5) and (6).

In general the action-based skinny encoding gives the most robust performance, simply because as the smallest in terms of both variables and clauses it is least likely to result in a formula that is too large to fit into main memory. (Satisfiability testing and virtual memory are an unhealthy combination.)

The single most important difference between blackbox and SATPLAN04 is the SAT solvers used. Blackbox included the original graphplan (non-translation based) search engine, the local-search SAT solver walksat (Selman, Kautz, & Cohen 1994), the forward-checking DPLL-based solver satz (Li & Anbulagan 1997), and the clause-learning DPLL-based solvers relsat (Bayardo & Schrag 1997) and zChaff (Moskewicz *et al.* 2001).

By contrast, SATPLAN04 uses a single highly optimized DPLL-based solver called “siege”, that was developed by Lawrence Ryan as part of his research at Simon Fraser University under the direction of Prof. David Mitchell. Linux binaries of siege can be downloaded from <http://www.cs.sfu.ca/~loryan/personal/>. We thank Lawrence Ryan for permission to incorporate siege in SATPLAN04.

Siege, like relsat and zChaff, performs clause-learning (that is, inferring new clauses at backtrack points), and like zChaff uses optimized “watched literal” data structures for managing large clause sets efficiently. Beyond that it appears to incorporate a number of other optimizations that make it particularly well-suited for the planning as satisfiability approach. In our initial informal tests siege signifi-

cantly outperformed all the other solvers mentioned above. Later this summer we will post detailed comparisons of the different SAT solvers on planning formulas on our planning as satisfiability web page, <http://www.cs.washington.edu/homes/kautz/blackbox/>.

The PDDL parser in SATPLAN04 is considerably more robust than the one in blackbox, but it does not yet handle any non-STRIPS features other than types, such as derived effects and conditional actions. We plan to extend SATPLAN04 to handle these and other features in time for the 2005 planning competition.

## References

- Bayardo, R. J. J., and Schrag, R. C. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 203–208.
- Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, 1636–1642.
- Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, 359–363. Wiley.
- Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, 1194–1201. AAAI Press. (Best Paper Award).
- Kautz, H., and Selman, B. 1999. Unifying sat-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 318–325. Morgan Kaufmann.
- Li, C. M., and Anbulagan. 1997. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 97)*, 366–371.
- Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient sat solver. In *39th Design Automation Conference*.
- Selman, B.; Kautz, H.; and Cohen, B. 1994. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, 337–343. AAAI Press.

# Tilsapa – Timed Initial Literals Using SAPA

Bharat Ranjan Kavuluri

[bharat@cs.iitm.ernet.iin](mailto:bharat@cs.iitm.ernet.iin)

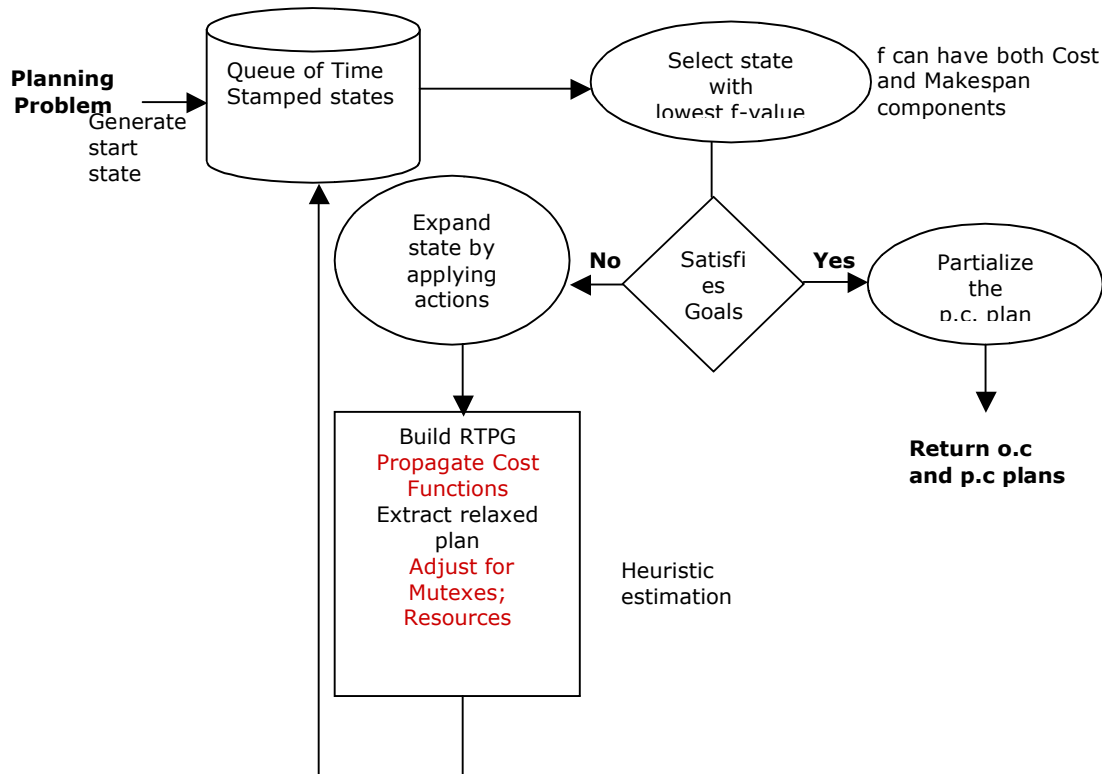
Senthil U

[you\\_yes@engineer.com](mailto:you_yes@engineer.com)

AIDB Lab  
IIT Madras  
Guindy, Chennai  
India- 600036

## System Abstract

This system is an offshoot of SAPA[1] developed by Binh Minh Do and Subbarao Khambampati. The following diagram represents the architecture of SAPA.



[1]Architecture diagram of SAPA

A time stamped state can be described as a quintuple  $S = (P, M, \Pi, Q, t)$  where

$P = \text{Set } \langle p_i, t_i \rangle$  of predicates  $p_i$  and the time of their last achievement  $t_i < t$ .

$M = \text{Set of functions representing resource values.}$

$\Pi = \text{Set of protected persistent conditions}$

$Q = \text{Queue of future events}$

$T = \text{Time stamp of } S$  [1]



- Timed Initial Literals are implemented using SAPA by the following method
  - Include them in the event queue at the outset (Plan request)
  - Include them in the predicate set P before the state is expanded by applying new actions
- Derived predicates can be introduced before any action is considered into the set of Predicates, which are valid for the current state.
- Constants are included with each plan request as initial predicates.

The major bottleneck is the heuristic computation and propagation of the cost where

- it is assumed that each predicate can only be caused by an action.
- the heuristic value of the state is a function of
  - Cost of the relaxed plan from this state to the goal state.
  - Makespan of the relaxed plan.

We are currently working on optimizing the cost propagation process after taking the timed initial literals into consideration. The system is still under implementation.

#### **References:**

[1] Do, M. and Kambhampati, S. (2003) "SAPA: A Multi-objective Metric Temporal Planner", Journal of Artificial Intelligence Research, Volume 20, pages 155-194.

# The Optop Planner

Drew McDermott

Yale University Computer Science Department  
P.O. Box 208285  
New Haven, CT 06520-8285  
drew.mcdermott@yale.edu

## Introduction

Optop<sup>1</sup> is an *estimated-regression* planner, meaning that it is a “state-space planner” that is guided by a heuristic measure of how close a situation is to satisfying a goal, and how good it is according to an objective function. Research on Optop is focused more on deep reasoning about situations and transitions than on raw performance.

Instead of talking about “state space,” I prefer to characterize the search space of Optop as the set of *plan prefixes*, that is, sequences of actions that are executable starting in the initial state. Such a sequence generates a unique situation, called the *current situation* for that prefix.

## Heuristic Search Using Estimated-Regression Graphs and Objective Functions

Optop decides which plan prefix to work on next using a heuristic inspired by *means-ends analysis* (Ernst & Newell 1969). For each plan prefix, it constructs a *regression-match graph* that is a simplified prediction of how that goal might be achieved starting in the current situation for a given plan prefix. The graph is constructed by *maxmatching* the goal against the current situation, which produces a substitution (called a *maximal match*) that binds the variables in the goal so as to make as many of its conjuncts true as possible. The remaining conjuncts, the *differences* left by the maxmatch, become subgoals. For each literal in differences, Optop finds all the actions, processes, or implications that could make it true. Each has some kind of precondition that is maxmatched against the current situation, giving further differences. As this process is continued, a tripartite graph emerges, each of whose nodes is of one of the following three types:

1. An *L-node*: A literal occurring as differences in a maxmatch.

2. An *effort-spec*: An L-node plus numerical constraints on its free variables. Numerical constraints can’t be handled by regression, but must be postponed and satisfied by a special numerical module at the appropriate time.
3. A *reduction*: A record of the application of a “regression method” to an effort-spec. A typical regression method corresponds to an action definition, and specifies sufficient conditions for that action to cause each of its possible effects. (Some of the other kinds are discussed below.) These conditions are maxmatched to derive a set of differences, each of which is an effort-spec in the graph.

Each effort-spec may have several reductions, and each reduction has a set of precondition effort-specs which are sufficient to ensure that the action, process, or implication associated with the reduction will cause the L-node of the effort-spec to be true. (Actually, reductions and maxmatches are cached on L-nodes; when an effort-spec for an L-node, is built, Optop copies the reductions, adds the numerical constraints if any, and verifies that they are satisfiable.)

L-nodes and effort-specs are “uniquified”; that is, if an equal L-node already exists, it is used instead of a new one being created. That means the regression-match graph for a planning problem tends to be much smaller than its situation space.

The graph yields a rough estimate of the difficulty of the problem, obtained by counting the actions in a subtree of the graph that is minimal in a sense explained in (McDermott 1996; 1999). However, many planning problems include a specification of an “objective function” to be minimized. Optop finds linearizations of the regression-match graph that then give rise to *plausible projections* of the rest of the plan. The result is a collection of feasible actions and speculative versions of the final situation that might follow from them, and Optop evaluates the objective function in those situations to produce estimates of the quality of alternative extensions of the current plan prefix (McDermott 2003).

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

<sup>1</sup>This looks like an acronym for something; ordered? operator? tops? How is it syllabified, as Opt-op or Op-top?

## Expressivity

In addition to actions, Optop can reason about autonomous processes, which run whenever their conditions are true without the need for planner intervention. The planner can plan to bring these into existence by making the condition true, or can take advantage of processes defined as part of the problem.

Optop can handle all of ADL (Pednault 1989), including universally-quantified preconditions. It uses the Screamer system (Siskind & McAllester 1993) to solve numerical constraints, especially those that arise in connection with predicting when processes will cause something to become true.

The reason for Optop's versatility is that its reasoning is closely tied to complete descriptions of situations, unlike partial-order planners (Weld 1994) and Graphplan-style planners (Blum & Furst 1997), which reason about goal-satisfaction links, mutual-exclusion relations, and the like without tying them to any particular situation. Generating the regression-match graph requires reasoning backward from the goal to the current situation, and can use any reasoning technique, domain-dependent or -independent, without worrying about enough information is known about that situation. (Of course, that is not the only consideration; Optop is no better than other Strips-style planners in doing regression involving geometrical reasoning.)

Once an action is chosen to explore, Optop typically generates a new current situation following that action. However, if autonomous processes are active, the next situation is the one that occurs when those processes cause a discrete change of some kind. Again, just about any computation that projects the sequelae of the current state of affairs is easy to exploit.

In addition to its heuristic evaluator, a planner must also have a search strategy. Optop uses best-first search as long as its heuristic is sharply differentiating among alternative plan prefixes. When too many accumulate that seem to be of about the same quality, it switches to a strategy of "hill climbing with random restarts." In this mode, it always extends the plan prefix with the action that looks the best locally; that is, if it has to choose among actions it with  $A_1, \dots, A_k$ , it picks an  $A_i$  that dominates  $A_1, \dots, A_n$ , without regard to previously generated possibilities. If it reaches a point where there is no feasible action that leads to a new situation, it makes a random choice among all the plan prefixes it has generated and resumes hill climbing from there.

## Changes for the Competition

To illustrate how easily changes are made to Optop, here's an account of recent changes to the system.<sup>2</sup>

The ability to handle universally-quantified preconditions was added to Optop for this year's IPC. An ordinary precondition set such as `(and (Q ?y) (P a ?y))`

is handled during maxmatching by finding values for `?y` that make either `(Q ?y)` or `(P a ?y)` true. The other precondition, with `?y` substituted away, becomes a *difference* to be reduced. Now suppose we have preconditions `(and (Q ?y) (forall (z) (if (R ?y) (P ?y z))))`. Suppose `?y=b` make `(Q ?y)` true. Then the remaining differences are all the literals whose unprovability produces a counterexample to the universal. A counterexample is an instance of `(and (R z) (not (P b z)))`, which can be produced by finding  $z$ 's such that `(R z)` is provable and `(P b z)` is not; each such `(P b z)` becomes a difference. Writing and plugging in the code for this mechanism was a relatively simple task.

Note that the maxmatcher must find values for `?y` before considering the universal. That's because there is no way to enumerate all the values  $y$  that make `(forall (z) (if (R y) (P y z)))` provable, or all those that make it unprovable. (Provability is used as a stand-in for truth, because PDDL relies on a closed-world assumption: if a proposition can't be proved, it is taken to be false.) The deductive system built-in to Optop distinguishes between queries with no answers and queries with an unknown number of answers that might be handled if more of their free variables are bound. This turns out to be a very useful feature with a variety of uses, one of which is to decide how to order preconditions during maximal matching.

For the competition, PDDL was extended in two further ways: with derived predicates and timed initial literals. Optop already had derived predicates, which it used in the following way: Suppose, in the previous example, there was an axiom<sup>3</sup>

```
(forall (x)
  (<- (Q ?y)
    (exists (v)
      (and (R v ?y) (R ?y v)))))
```

The existence of this axiom gives the maxmatcher an extra degree of freedom. Instead of having to classify `(Q a)` as a difference, it can also find a  $v$  such that `(R v a)` and make `(R a v)` a difference. The term *derived predicate* is just another name for a predicate defined by a single backward-chaining axiom.

Unfortunately, expanding axioms this way is not a good idea unless the axioms are *stratified*, meaning that there is no path from a predicate to itself through the axioms in question. To handle those correctly, we have to cope with the recursion by moving it out to the level of the regression-match graph. That is, an unstratified axiom gives rise to a different kind of regression method, in which the conditions lead immediately to a conclusion with no action or process intervening. For example, the unstratified axiom

```
(forall (x y)
  (<- (above ?x ?y)
    (exists (w)
```

<sup>2</sup>Optop is written in Lisp; I can't imagine how it could evolve so quickly if it were written in any other language.

<sup>3</sup>The "`<-`" indicates that the implication is to be used for backward chaining.

(and (above ?x w)  
(above w ?y))))))

can be used to reduce an L-node (above a e) to (and (above a ?w) (above ?w e)), which, after maxmatching, yields subgoal nodes such as (above b e) (if (above a b) is true in the current situation). An L-node can easily occur as a sub-sub-...-node of itself, but such cycles are simply ignored when the regression-match graph is used to produce and evaluate extensions of the current plan prefix.

## Performance

As shown in (McDermott 1999), although Optop spends more time per search state than other planners, in some domains it explores so few states that its run times are comparable to highly optimized systems. On “well-behaved” domains, its run times grow polynomially with problem size.

There is a price to be paid for Optop’s flexibility. The relaxed search space embodied in the regression-match graph neglects destructive interactions among actions (Bonet, Loerincs, & Geffner 1997; Bonet & Geffner 2001). This neglect makes it difficult to solve problems in domains in which a crucial condition can be irreversibly deleted without that being discovered until several more actions have been added to the plan. (The classic example is the “Rockets” domain of (Blum & Furst 1995).) On the other hand, realistic domains are often more forgiving, and allow problems to be broken into loosely coupled subproblems that can be solved by the sort of hill climbing described above.

## Future Plans

My current research goal is to add hierarchical and contingency planning to Optop. The former requires augmenting search states with information about hierarchical plans (i.e., canned plans from a library) that are in progress. With this addition, the regression-match graph will be built to handle posted but unsatisfied goals from the current hierarchical plan — the *script*. An action that is already in the script will not normally be proposed, unless a new instance is needed in order to achieve a precondition of some other step.

Contingency planning is mainly a matter of running the planner for various alternative scenarios. The mechanics are easy; the hard part is deciding when to stop exploring contingencies.

## References

- Blum, A. L., and Furst, M. L. 1995. Fast planning through planning graph analysis. In *Proc. Ijcai*, volume 14, 1636–1642.
- Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 1–2 90:279–298.
- Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2).

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A fast and robust action selection mechanism for planning. In *Proc. AAAI-97*.

Ernst, G. W., and Newell, A. 1969. *GPS: A Case Study in Generality and Problem Solving*. Academic Press.

McDermott, D. 1996. A Heuristic Estimator for Means-ends Analysis in Planning. In *Proc. International Conference on AI Planning Systems*, 142–149.

McDermott, D. 1999. Using Regression-match Graphs to Control Search in Planning. *Artificial Intelligence* 109(1-2):111–159.

McDermott, D. 2003. Reasoning about autonomous processes in an estimated-regression planner. In *Proc. Int’l Conf. on Automated Planning and Scheduling*.

Pednault, E. P. D. 1989. Adl: Exploring the middle ground between Strips and the situation calculus. In *Proc. Conf. on Knowledge Representation and Reasoning*, volume 1, 324–332.

Siskind, J. M., and McAllester, D. A. 1993. Non-deterministic Lisp as a substrate for constraint logic programming. In *Proc. AAAI 1993*, 133–138.

Weld, D. 1994. An introduction to least-commitment planning. *AI Magazine*.

# Combining Backward-Chaining with Forward-Chaining AI Search

Eric Parker

eric@semsyn.com

SemSyn is an automatic plan synthesis algorithm that endeavors to fulfill the requirements of flexible, industrial-strength, next-generation AI planning. Historically, AI planning systems have not been viewed as practical because users have had to be skilled artificial intelligence practitioners. This is in part due to the fact that systems built to solve large-scale, real-world problems traditionally rely on optimisation and/or heuristic procedures. Further difficulties with such systems are that optimisation procedures are usually tailored around specific types of problems, and that heuristic procedures are not guaranteed to find a solution. The former approach produces planning systems that are inflexible. The latter approach produces planning systems that are unsuitable for industrial settings that require critical systems.

SemSyn, on the other hand, performs an exhaustive search, thereby retaining both completeness and flexibility. The algorithm combines well-known forward-chaining search (FCS) and backward-chaining search (BCS) strategies from the AI literature (e.g. [5]). That is, the children-generation function of FCS consists of producing the domain actions that are applicable in the current *state*, while the children-generation function of BCS consists of producing the domain actions that are applicable to the current set of *subgoals*. Intuitively, combining the approaches seems to be the right move, since a desirable outcome is that some *subgoals* (namely, the top-level goals) are satisfied in some *state*. In any case, FCS and BCS separately share the common fate of combinatorial explosion, and SemSyn hopes to play the strengths of one against the weaknesses of the other (in the spirit of [4]). This is done by using a generalised BCS to compute the causal link information, and by using the FCS states to impose a total order on (some subset of) the causal links. The causal links computation must be efficient enough so as not to outweigh the benefit of their use.

## Classical Backward-Chaining Search

SemSyn implements BCS in a breadth-first manner and employs a sideways-information-passing technique that provides an upper bound on the number of actions at each level of the search. The SemSyn approach can be better understood in relation to the classical BCS approach. The root of the classical BCS search tree consists of the top-

level problem goals. The root's children are those domain actions that both achieve some top-level goal and don't delete any of the top-level goals. Domain actions that meet these requirements are said to be applicable to the top-level goals, or more generally, they are said to be applicable to the current set of subgoals. The current set of subgoals for each child is computed from its parent's subgoals by regressing the parent subgoals through the child [7]. The children-generation function is then re-applied to each of these nodes to produce the root's grandchildren, and so on. When BCS is implemented in a breadth-first manner it builds action sequences of increasing length, which provides the opportunity to apply sideways-information-passing techniques [1].

## Generalised Backward-Chaining Search

SemSyn's backward-chaining search (SBCS) differs from BCS in two important ways: 1. Instead of having a single root, the root level of the SBCS search tree (in fact, a graph) has one node for each top-level goal. The current set of subgoals for each of these "root" nodes consists only of the node's top-level goal. Put differently, SBCS builds partial plans, whereas BCS builds plans. Since partial plans are, hopefully, shorter than plans, the total amount of work is sometimes reduced. 2. SBCS tries to pass information between partial plans of equal length. The strategy relies on the fact that the same domain action can be applied to more than one set of subgoals at each level of the graph. For every level  $L$  of the graph, and for every domain action, if the action is applicable to  $n$  subgoal sets of  $L$ , then create one child having two sets of subgoals. One set of subgoals, *u-subgoals*, is the union of all of the  $n$  subgoal sets, and the other set of subgoals, *x-subgoals*, is the intersection of all of the  $n$  subgoal sets. Note that since all of the  $n$  subgoal sets are computed by regression through the same action, *x-subgoals* may only be empty for a domain action that has no precondition. Note also that not considering secondary preconditions during the regression may lead to incompleteness. For example, this occurs when an action  $A$  with no precondition and a single conditional effect  $E$  has an instantiated predicate as the antecedent of  $E$ . In this case,  $A$  is functionally

equivalent to an action  $B$ , where the precondition of  $B$  is the antecedent of  $E$ , and  $B$ 's effect is the consequent of  $E$ .

Next, we generalise what it means for a domain action to be applicable to a set of subgoals, since we now have a double of subgoals. A domain action is applicable to a subgoal double ( $u$ -subgoals,  $x$ -subgoals) if it both achieves some subgoal in  $u$ -subgoals and doesn't delete any subgoal in  $x$ -subgoals. Because of the generalization it is possible to generate more children from a particular node than the usual way, but the generalisation also has the special property that it puts an upper bound on the number of children generated for a particular level. In the worst case, each level of the graph contains no more nodes than there are domain actions (in the spirit of [2]). In and of itself, the generalisation of subgoal sets is admittedly naïve. However, on the whole, it is instructive to try to convince oneself that the SBCS graph contains all of the causal links, and that no solutions will be lost.

### Goal-Directed Forward-Chaining Search

SemSyn's forward-chaining search (SFCS) is relegated to the task of searching the SBCS causal links, in effect assembling partial plans into plans. The children-generation function of SFCS differs from that of FCS in that the candidates are not chosen from the entire set of domain actions, but rather are constrained to be only those domain actions appearing in an appropriate causal link entry. In particular, if none of the domain actions achieve any of the top-level goals, then SFCS will terminate immediately without generating any children, whereas FCS in the worst case degenerates into a blind enumeration of all action sequences possible from the initial situation. SFCS alone has no more pruning ability than FCS. The research effort thus far has just been to integrate BCS and FCS, and to evaluate the usefulness of doing so. It is hoped that SemSyn will eventually provide a useful tool for further study.

### Putting It All Together

Finally, next-generation planning systems will need to interact with their human users. This was one of the driving motivations for the research community's move from an automatic to an automated paradigm. We posit that automatic algorithms can still be useful as sub-modules to the more encompassing automated systems. Moreover, SBCS and SFCS have human-understandable and intuitive children-generation functions, as do BCS and FCS, so it becomes alluring the possibility to allow the user to view and to manipulate the search's internal data structures - they are simply plan fragments! It is our thesis that user's of automatic planning systems are freed from planning concerns, and are able to fully concentrate on the domain modeling aspects of their applications.

The initial testing phase is being carried out in cooperation with SemSyn's participation in the 4<sup>th</sup> International

Planning Competition (IPC), hosted at the 2004 International Conference on Automated Planning and Scheduling. The IPC series has developed a formidable testbed, and a rigorous evaluation of the results is forthcoming.

SemSyn's preliminary results appear satisfactory insofar as it is able to solve problems from a variety of domains. However, the algorithm has trouble with domains that have relatively little variation in the domain operators. This is because the traditional wisdom of the research culture is to design a sequence of problems of increasing difficulty in an artificial way, by increasing the number of actions that can be instantiated from a few operators, i.e. by increasing the number of predicates the operators have at their disposal. Conversely, in SemSyn's view, the predicates are akin to database tuples. This means that it is the user's responsibility to model the domain in such a way as the predicate space can be efficiently explored. Indeed, it is possible to write database queries that don't terminate, yet people routinely use Database Management Systems as an integral part of their overall information systems. Analogously, SemSyn's goal is to separate the planning aspects from the domain modeling activities, and to devote its effort to the task of planning - that is, the efficient construction of plans based upon knowledge encoded in the domain operators themselves, regardless of instantiated actions.

### References

1. Beeri, C. and Ramakrishnan, R. 1991, "On The Power of Magic". In *Journal of Logic Programming*, 10(3):255-299.
2. Blum, A. and Furst, M.L. 1995, "Fast Planning Through Planning Graph Analysis". In *Proc. International Joint Conference on Artificial Intelligence*, pp. 1636-1642.
3. Fikes, R.E. and Nilsson, N.J. 1971, "STRIPS: a new approach to the application of theorem proving to problem solving". *Artificial Intelligence*, 2(3-4):189-208.
4. Fuchs, D. and Fuchs, M. 1999, "Cooperation between Top-Down and Bottom-Up Theorem Provers". *Journal of Artificial Intelligence Research*, 10:169-198.
5. Kambhampati, S. 1997, "Refinement Planning as a Unifying Framework for Plan Synthesis". *AI Magazine*, 18(2):67-97.
6. Lifschitz, V. 1986, "On the semantics of STRIPS". In *Proc. Reasoning about Actions and Plans*, pp. 1-9. Morgan Kaufmann.
7. McDermott, D. 1996, "The Current State of AI Planning Research". Invited paper, 9<sup>th</sup> Intl. Conf. on Industrial and Engineering Applications of AI and Expert Systems, pp. 25-34.

# P-MEP: Parallel More Expressive Planner

Javier Sanchez (\*), Minh Tang & Amol D. Mali

Electrical Engineering & Computer Science,  
University of Wisconsin, Milwaukee, WI 53211,

(\*) EDESA S.A, Cra 18 86A-14, Bogota, Colombia, { javier, minh, mali }@uwm.edu

P-MEP [10] is a forward state-space planner that performs weighted A\* style search. It allows a user to choose the heuristic to be used and the weight in weighted A\*. It has relevance analysis as a preprocessing technique to control search. P-MEP uses the notions of referenced and updated variables to detect equivalent states to control search. The key ideas in P-MEP are the use of mutual exclusion relations (mutexes) in the computation of relaxed plans and the use of intervals of relaxed values. The notion of relaxed intervals in P-MEP is inspired by relaxations in Sapa and Metric-FF. The relaxed intervals allow P-MEP to handle expressions containing  $+$ ,  $-$ ,  $/$ ,  $*$ , exponentiation,  $=$ ,  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $\wedge$ ,  $\vee$ ,  $\neg$ ,  $\rightarrow$ ,  $\forall$ , and  $\exists$ . Relaxed interval of a variable contains the minimum and maximum relaxed values of the variable. Relaxed intervals are useful in several ways. They allow P-MEP to handle decrease effects of actions and numerical preconditions in the computation of relaxed plans. P-MEP uses the relaxed intervals to check if numerical preconditions are achievable in a relaxed fashion. Relaxed intervals also allow P-MEP to handle linear and non-linear expressions in the goal of a planning problem. The relaxed intervals also allow an easy detection of whether a numerical goal or numerical subgoal is achievable in a relaxed fashion.

P-MEP creates all ground instances of operators before search begins. If the domain description does not specify operator durations, P-MEP assumes that they are all unity and treats the domain as a temporal domain.

**World State:** P-MEP treats propositions and ground predicates as numerical variables with domain  $\{0, 1\}$ . A world state  $S$  in a node  $N$  in the search tree of P-MEP is the tuple  $\langle V, a, t \rangle$ , where  $t$  is time stamp of  $S$ ,  $V$  is the set of numerical variable-value pairs, and  $a$  is the action applied to generate  $N$ . Time stamp of a world state  $S$  is the earliest time at which an action can be applied in  $S$ . P-MEP applies actions so that they start at the earliest possible times.

**Applicable Actions:** The conditions for applicability of an action  $o'$  in a node  $n$  with world state  $S$  are: (i) all preconditions of  $o'$  that need to be true at its start or end points must be satisfied by  $V$  of  $S$ , (ii) all preconditions of  $o'$  that need to be true over its entire interval must also be satisfied by  $V$  of  $S$ , and (iii) effects of  $o'$  do not change the value of any boolean or numerical variable in the preconditions or effects of any action that  $o'$  overlaps with and vice versa.

**Statically Mutex Actions:** For each action  $o'$ , P-MEP computes the set of variables updated by the action and the set of variables that are referenced by the action. These sets are denoted by  $U(o')$  and  $R(o')$  respectively. These sets contain predicates and discrete variables. As an example, let the preconditions of  $o'$  be  $a, b$  and  $c > 200, e < 100$ , where  $a$  and  $b$  are propositions and  $c, e$  are discrete variables. Let the effects of this action be  $\neg a, d, c = (c + e - 50)$ , where  $d$  is a proposition. In this case,  $R(o') = \{a, b, c, e\}$  and  $U(o') = \{d, a, c\}$ . In general, any variable that appears in preconditions or effects of an action and which is not updated by the action is a referenced variable for that action. Two actions  $o_i$  and  $o_j$  are statically mutually exclusive in P-MEP if one or more of the following three conditions are satisfied: (i)  $(R(o_i) \cap U(o_j)) \neq \phi$ , (ii)  $(R(o_j) \cap U(o_i)) \neq \phi$ , (iii)  $(U(o_i) \cap U(o_j)) \neq \phi$ . Statically mutually exclusive actions are permanently mutually exclusive and cannot overlap.

**Equivalent States:**  $R$  is the set of variables referenced by one or more actions and  $U$  is the set of variables updated by one or more actions. Two states  $s$  and  $s'$  are equivalent if the values of all variables in  $R \cap U$  are same in the  $V$  component of  $s$  and  $s'$ . This definition of equivalent states allows P-MEP to control search by not visiting multiple world states that differ only in the value of variables from  $U - R$ . The variables in  $U - R$  do not affect the applicability of actions. Consider the variable total-fuel-consumed denoting the total fuel consumed by a partial plan, in transportation logistics domain. This variable is not relevant to

achieving any precondition of any action. This variable belongs to  $U$  but not to  $R$ . This variable can have infinite non-negative values since infinite flights are possible. By not considering such variables in the state equivalence test, P-MEP controls the size of its search tree.

**Search Algorithm:** P-MEP conducts forward state-space search in weighted A\* style. The weighted variant of A\* uses the following path cost equation  $f(n) = (1 - w) * g(n) + w * h(n)$ ,  $0 \leq w \leq 1$ , where  $g(n)$  represents the cost of the path from the root node to node  $n$ , and the  $h(n)$  represents the estimate of the cost of the cheapest path from  $n$  to goal. In P-MEP, the nodes in fringe are sorted according to value of the  $f$  function. The node with lowest  $f()$  value is expanded first. If multiple nodes have the same value of  $f$ , then the node with lowest depth is expanded first. If nodes with the lowest value of  $f$  have the same depth, then the node that is generated earlier is expanded first. A node is generated by applying only one action. Since multiple actions may have the same starting time, concurrency is possible. An action  $o'$  can start during the interval of other actions that are not statically mutex with  $o'$ , making concurrency possible. P-MEP terminates when there is a node  $n$  such that every subgoal is true in the  $V$  component of the world state in  $n$ .

**Relaxed planning graph (RPG):** The notion of RPG was introduced in FF planner [6]. We denote the goal of a planning problem by  $G$  in the rest of the paper. A subgoal from  $G$  is an expression from  $G$ . An RPG is constructed by FF assuming that the delete effect lists of actions are empty. The notion of proposition level is replaced by the notion of variable level, in order to construct an RPG for more expressive domains.  $i$  th action level occurs between  $i$  th variable level and  $i + 1$  the variable level. P-MEP constructs a serial relaxed planning graph. P-MEP stores an interval bounded by maximum and minimum relaxed values for each variable in each variable level. A variable level is a set of  $\langle v, [min(v), max(v)] \rangle$  tuples, where  $min(v)$  and  $max(v)$  are minimum and maximum relaxed values of variable  $v$ . The size of an interval is monotonically increasing. The interval for a variable  $v'$  in  $i$  th variable level is obtained by updating its interval in the  $i - 1$  the variable level with the effects of the action in the  $i - 1$  th action level. For example, let the value of variable  $v_1$  in the world state of a node  $n$  be 4. Then the interval for  $v_1$  in the first variable level in the RPG at  $n$  is  $[4, 4]$ . If an action increasing  $v_1$  by 10 is included in the first action level of the RPG, then the interval for  $v_1$  in second variable level is  $[4, 14]$ . If an action decreasing  $v_1$  by 20 is then included in the second action level of RPG, the inter-

val for  $v_1$  in the third variable level is  $[-16, 14]$ . If an action assigning 5 to  $v_1$  is then included in the next action level of RPG, the interval for  $v_1$  in the fourth variable level is still  $[-16, 14]$ , since 5 is in the interval  $[-16, 14]$ . The intervals for variables make it easy to compute relaxed intervals for expressions and check if the expressions are satisfied in a relaxed fashion in a variable level. For example, let the relaxed intervals for variables  $v_1, v_2$  be  $[-3, 5]$  and  $[-4, 8]$ . The interval of  $(v_1 + v_2)$  is  $[-7, 13]$ . The intervals for  $v_1 * v_2$ ,  $\frac{v_1}{v_2}$ , and  $v_1 - v_2$  are found in a similar fashion. The intervals of complex arithmetic expressions are found using intervals of individual variables and operator precedence. For example, the relaxed interval for  $v_1 * v_2 * v_3 * v_4$  is found from relaxed intervals of  $v_1 * v_2 * v_3$  and  $v_4$ . The interval of  $v_1 * v_2 * v_3$  is found from the intervals of  $v_1 * v_2$  and  $v_3$ . The intervals for variables and expressions can be considered as the intervals of relaxed values. This is because though P-MEP considers add, delete effects, increase, assign and decrease effects in computing intervals, it ignores the interactions between actions. So some of the values in the intervals may be impossible to achieve.

Intervals of expressions in preconditions or goal are found only to test if preconditions or goal are achieved in a relaxed fashion. Intervals for expressions make it easy to check if actions are applicable in RPG and if goal is true in RPG. For example, the precondition or subgoal  $(v_1 + v_2) = 50$  is true in RPG if 50 lies in the interval of  $v_1 + v_2$ . Similarly, the expression/subgoal  $v_1 < v_2$  is satisfied in a variable level in the RPG if  $min(v_1) < max(v_2)$  is satisfied in the variable level.

P-MEP constructs RPG for a node by applying actions in forward direction and by computing action and variable levels, until the intervals of variables satisfy all expressions in the goal in some variable level or no variable's interval changes, whichever occurs earlier. If some subgoal is not achieved in the RPG of a node  $n$ , P-MEP sets  $h(n)$  to  $\infty$  and keeps the node in the priority queue.  $n$  is expanded after all states with finite  $h()$  values are expanded.

**Relaxed plan:** Relaxed plans are used to compute  $h()$  values for nodes by P-MEP, like Metric-FF [7], Sapa [2]. Relaxed plan for a node  $n$  is found by P-MEP in two phases. In first phase, it removes irrelevant actions from the RPG of  $n$ . This removal leaves a subgraph of RPG with gaps (some action levels are empty). In the second phase, P-MEP converts this subgraph into a relaxed temporal plan by pushing actions back to the earliest possible time, ensuring that statically mutex actions do not overlap. P-MEP considers action durations only in the second phase.

The relaxed plan found by phase 1 is serial. It



PLN	PL	NV	TD	NPG	NGPG	CE	Q	DPG
Sapa	3	Y	Y					
LPG	3	Y	Y					
MIPS	1, 2, 3	Y	Y		Y		Y	
TP4	3	Y	Y					
MFF	1, 2	Y		Y	Y	Y	Y	Y
FF	1				Y	Y	Y	Y
VHPOP	1, 3	Y			Y	Y	Y	Y
P-MEP	1, 2, 3	Y	Y	Y	Y	Y	Y	Y

Figure 1: Expressiveness features handled by various planners from 2002 planning competition. Y: Yes (handled).

is parallelized in phase 2 because temporal planning problems generally involve makespan minimization. Makespan of parallelized relaxed plan of node  $n$  can be a better estimate of the makespan of the optimal plan that achieves the goal from node  $n$ , than the makespan of serial relaxed plan. The estimates of the makespan of optimal plan from  $n$  can be better if statically mutex actions do not overlap in the relaxed plan at  $n$ . Hence overlap of statically mutex actions is avoided in the parallel relaxed plan.

**Supported Domain Features:** The domain features supported by P-MEP and seven other planners that participated in the international planning competition in 2002 are shown in Table 1. TP4 and VHPOP are described in [5] and [8] respectively. The acronyms in this table have the following meanings: PLN: Planner, MFF: Metric-FF, PL: PDDL level, NV: Numeric variables, TD: Time Durations, NPG: Numerical preconditions and goal, NGPG: Negated preconditions and goal, CE: Conditional Effects, Q: Quantifiers, DPG: Disjunctive preconditions and Goal. PDDL is planning domain description language. PDDL 2.1 level 1 includes STRIPS and ADL. PDDL 2.1 level 2 is an augmentation of PDDL 2.1 level 1 with numeric variables. PDDL 2.1 level 3 is an augmentation of PDDL 2.1 level 2 with time. PDDL levels partially/fully handled by various planners are also shown in Table 1. P-MEP is the only planner that handles all domain features in Table 1. The most recent version of MIPS does handle ADL.

**Relevance Analysis:** This is used as a preprocessing technique to reduce the number of actions used in search. This technique is similar to relevance analysis in [9]. P-MEP gives a user an option to use relevance analysis. P-MEP constructs an extended and serial relaxed planning graph (ESRPG) for root node by applying actions in forward direction, as a part of relevance analysis. The RPG is extended because its growth may be continued even after all subgoals are achieved in a relaxed fashion in some variable level. P-MEP does not check for the achievement of subgoals

in the variable levels when it constructs the ESRPG. The construction of ESRPG stops when no new action is applicable and then the set of actions in various action levels in the ESRPG is returned as the relevant actions' set.

**Heuristics:** P-MEP allows user to choose a heuristic from the following four heuristics: Cost, Makespan, Sum duration and Actions. The heuristics are not new, but the actual heuristic values and plans differ due to different method of computing relaxed plans in P-MEP.

- [1] A. Blum and M. Furst, Fast Planning through planning graph analysis, . Artificial Intelligence, Vol.90(1-2), 1997, pp. 281-300.
- [2] M. Do and S. Kambhampati, SAPA: A multi-objective metric temporal planner, JAIR 20, 2003, pp. 155-194.
- [3] S. Edelkamp, Taming numbers and durations in the model checking integrated planning system, JAIR 20, 2003, pp. 195-238.
- [4] A. Gerevini, A. Saetti, and I. Serina, Planning through stochastic local search and temporal action graphs in LPG, JAIR 20, 2003, pp. 239-290.
- [5] P. Haslum and H. Geffner, Heuristic planning with time and resources, Proceedings of the European Conference on Planning, 2001.
- [6] J. Hoffmann and B. Nebel, FF: The FF Planning System: Fast plan generation through heuristic search, Journal of Artificial Intelligence Research, Vol. 14, 2001.
- [7] J. Hoffmann, The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric state variables, JAIR 20, 2003, pp. 291-341.
- [8] H. Younes and R. Simmons, VHPOP: Versatile Heuristic Partial-order Planning, JAIR 20, 2003, pp. 405-430.
- [9] J. Hoffmann and B. Nebel, RIFO Revisited: Detecting relaxed irrelevance, Proceedings of ECP, 2001.
- [10] J. Sanchez, Integrating efficiency and expressiveness in planning, M.S thesis, Computer science, University of Wisconsin, Milwaukee, April 2003.

# The YAHSP planning system: Forward heuristic search with lookahead plans analysis

Vincent Vidal

CRIL - Université d'Artois  
rue de l'Université - SP 16  
62307 Lens, France  
vidal@cril.univ-artois.fr

## Introduction

Planning as heuristic search has proven to be a successful framework for STRIPS non-optimal planning, since the advent of planners capable to outperform in most of the classical benchmarks the previous state-of-the-art planners Graphplan (Blum & Furst 1997), Blackbox (Kautz & Selman 1999), IPP (Koehler *et al.* 1997), STAN (Long & Fox 1999), LCGP (Cayrol, Régnier, & Vidal 2001), ... Although these planners (except LCGP) compute optimal parallel plans, which is not exactly the same purpose as non-optimal planning, they also offer no optimality guarantee concerning plan length in number of actions.

The planning as heuristic search framework indeed lead to some of the most efficient planners, as demonstrated in the two previous editions of the International Planning Competition with planners such as HSP2 (Bonet & Geffner 2001), FF (Hoffmann & Nebel 2001) and AltAlt (Nguyen, Kambhampati, & Nigenda 2002). FF was in particular awarded for outstanding performance at the 2<sup>nd</sup> International Planning Competition and was generally the top performer planner in the STRIPS track of the 3<sup>rd</sup> International Planning Competition.

The YAHSP planning system ("Yet Another Heuristic Search Planner", more details in (Vidal 2004)) extends a technique introduced in the FF planning system (Hoffmann & Nebel 2001) for calculating the heuristic, based on the extraction of a solution from a planning graph computed for the relaxed problem obtained by ignoring deletes of actions. It can be performed in polynomial time and space, and the length in number of actions of the relaxed plan extracted from the planning graph represents the heuristic value of the evaluated state. This heuristic is used in a forward-chaining search algorithm to evaluate each encountered state.

We introduce a novel way for extracting information from the computation of the heuristic, by considering the high quality of the relaxed plans extracted by the heuristic function in numerous domains. Indeed, the beginning of these plans can often be extended to solution plans of the initial problem, and there are often a lot of other actions from these plans that can effectively be used in a solution plan. YAHSP uses an algorithm for combining some actions from each re-

laxed plan, in order to find the beginning of a valid plan that can lead to a reachable state. Thanks to the quality of the extracted relaxed plans, these states will frequently bring us closer to a solution state. The lookahead states thus calculated are then added to the list of nodes that can be chosen to be expanded by increasing order of the numerical value of the heuristic. The best strategy we (empirically) found is to use as much actions as possible from each relaxed plan and to perform the computation of lookahead states as often as possible.

This lookahead strategy can be used in different search algorithms. We propose a modification of a classical best-first search algorithm in a way that preserves completeness. Indeed, it simply consists in augmenting the list of nodes to be expanded (the open list) with some new nodes computed by the lookahead algorithm. The branching factor is slightly increased, but the performances are generally better and completeness is not affected.

Our experimental evaluation of the use of this lookahead strategy in a complete best-first search algorithm demonstrates that in numerous planning benchmark domains, the improvement of the performance in terms of running time and size of problems that can be handled have been drastically improved (cf. (Vidal 2004)).

## Computing and using lookahead states and plans

A *state* is a finite set of ground atomic formulas (i.e. without any variable symbol) also called *fluents*. *Actions* are classical STRIPS actions. Let  $a$  be an action;  $Prec(a)$ ,  $Add(a)$  and  $Del(a)$  are fluent sets and respectively denote the preconditions, add effects, and del effects of  $a$ . A *planning problem* is a triple  $\langle O, I, G \rangle$  where  $O$  is a set of actions,  $I$  is a set of fluents denoting the initial state and  $G$  is a set of fluents denoting the goal. A *plan* is a sequence of actions. The *application* of an action  $a$  on a state  $S$  (noted  $S \uparrow a$ ) is possible if  $Prec(a) \subseteq S$  and the resulting state is defined by  $S \uparrow a = (S \setminus Del(a)) \cup Add(a)$ . Let  $P = \langle a_1, a_2, \dots, a_n \rangle$  be a plan.  $P$  is *valid* for a state  $S$  if  $a_1$  is applicable on  $S$  and leads to a state  $S_1$ ,  $a_2$  is applicable on  $S_1$  and leads to  $S_2$ , ...,  $a_n$  is applicable on  $S_{n-1}$  and leads to  $S_n$ . In that case,  $S_n$  is said to be *reachable* from  $S$  for  $P$  and  $P$  is a *solution plan* if  $G \subseteq S_n$ . *First(P)* and *Rest(P)* respec-

tively denote the first action of  $P$  ( $a_1$  here) and  $P$  without the first action ( $\langle a_2, \dots, a_n \rangle$  here). Let  $P' = \langle b_1, \dots, b_m \rangle$  be another plan. The *concatenation* of  $P$  and  $P'$  (denoted by  $P \oplus P'$ ) is defined by  $P \oplus P' = \langle a_1, \dots, a_n, b_1, \dots, b_m \rangle$ .

### Principle and use of lookahead plans

In classical forward state-space search algorithms, a node in the search graph represents a planning state and an arc starting from that node represents the application of one action to this state, that leads to a new state. In order to ensure completeness, all actions that can be applied to one state must be considered. The order in which these states will then be considered for development depends on the overall search strategy: depth-first, breadth-first, best-first. . .

Let us now imagine that for each evaluated state  $S$ , we knew a valid plan  $P$  that could be applied to  $S$  and would lead to a state closer to the goal than the direct descendants of  $S$  (or estimated as such, thanks to some heuristic evaluation). It could then be interesting to apply  $P$  to  $S$ , and use the resulting state  $S'$  as a new node in the search. This state could be simply considered as a new descendant of  $S$ .

We have then two kinds of arcs in the search graph: the ones that come from the direct application of an action to a state, and the ones that come from the application of a valid plan to a state  $S$  and lead to a state  $S'$  reachable from  $S$ . We will call such states *lookahead states*, as they are computed by the application of a plan to a node  $S$  but are considered in the search tree as direct descendants of  $S$ . Nodes created for lookahead states will be called *lookahead nodes*. Plans labeling arcs that lead to lookahead nodes will be called *lookahead plans*. Once a goal state is found, the solution plan is then the concatenation of single actions for arcs leading to classical nodes and lookahead plans for the arcs leading to lookahead nodes.

Completeness and correctness of search algorithms are preserved by this process, because no information is lost: all actions that can be applied to a state are still considered, and because the nodes that are added by lookahead plans are reachable from the states they are connected to. The only modification is the addition of new nodes, corresponding to states that can be reached from the initial state.

### Computing relaxed plans

The determination of an heuristic value for each state as performed in the FF planner offers a way to compute such lookahead plans. FF creates a planning graph for each encountered state  $S$ , using the relaxed problem obtained by ignoring deletes of actions and using  $S$  as initial state. A relaxed plan is then extracted in polynomial time and space from this planning graph. The length in number of actions of the relaxed plan corresponds to the heuristic evaluation of the state for which it is calculated. Generally, the relaxed plan for a state  $S$  is not valid for  $S$ , as deletes of actions are ignored during its computation: negative interactions between actions are not considered, so an action can delete a goal or a fluent needed as a precondition by some actions that follow it in the relaxed plan. But actions of the relaxed plans are used because they produce fluents that can be interesting to obtain the goals, so some actions of these plans

can possibly be interesting to compute the solution plan of the problem. In numerous benchmark domains, we can observe that relaxed plans have a very good quality because they contain a lot of actions that belong to solution plans.

The computation of relaxed plans in YAHSP works closely as in FF, with one notable difference which holds in the way actions are added to the relaxed plan. In FF, actions are arranged in the order they get selected. We found useful to use the following algorithm. Let  $a$  be an action, and  $\langle a_1, a_2, \dots, a_n \rangle$  be a relaxed plan. All actions in the relaxed plan are chosen in order to produce a subgoal in the relaxed planning graph at a given level, which is either a problem goal or a precondition of an action of the relaxed plan.  $a$  is ordered after  $a_1$  iff:

- the level of the subgoal  $a$  was selected to satisfy is strictly greater than the level of the subgoal  $a_1$  was selected to satisfy, or
- these levels are equal, and either  $a$  deletes a precondition of  $a_1$  or  $a_1$  does not delete a precondition of  $a$ .

In that case, the same process continues between  $a$  and  $a_2$ , and so on with all actions in the plan. Otherwise,  $a$  is placed before  $a_1$ .

### Computing lookahead plans

The algorithm for computing lookahead plans (cf. Figure 1) takes as input the current planning state  $S$ , and the relaxed plan  $RP$  that has been computed by the heuristic function. Several strategies can be imagined: searching plans with a limited number of actions, returning several possible plans, etc. From our experiments, the best strategy we found is to search one plan, containing as most actions as possible from the relaxed plan. One improvement we made to that process is the following. When no action of  $RP$  can be applied, we replace one of its action  $a$  by an action  $a'$  taken from the global set of actions  $O$ , such that  $a'$ :

- does not belong to  $RP$ ,
- is applicable in the current lookahead state  $S'$ ,
- produces at least one add effect  $f$  of  $a$  such that  $f$  is a precondition of another action in  $RP$  and  $f$  does not belong to  $S'$ .

At first, we enter in a loop that stops if no action can be found or all actions of  $RP$  have been used. Inside this loop, there are two parts: one for selecting actions from  $RP$ , and another one for replacing an action of  $RP$  by another action in case of failure in the first part.

In the first part, actions of  $RP$  are observed in turn, in the order they are present in the sequence. Each time an action  $a$  is applicable in  $S$ , we add  $a$  to the end of the lookahead plan and update  $S$  by applying  $a$  to it (removing deletes of  $a$  and adding its add effects). Actions that cannot be applied are kept in a new relaxed plan called *failed* in the order they get selected. If at least one action has been found to be applicable, when all actions of  $RP$  have been tried, the second part is not used (this is controlled by the boolean *continue*). The relaxed plan  $RP$  is overwritten with *failed* and the process is repeated until  $RP$  is empty or no action can be found.

```

function lookahead ( $S, RP$ ) /*  $S$ : state,  $RP$ : relaxed plan */
  let  $plan = \langle \rangle$ ;
  let  $failed = \langle \rangle$ ;
  let  $continue = true$ ;
  while  $continue \wedge RP \neq \langle \rangle$  do
     $continue \leftarrow false$ ;
    forall  $i \in [1, n]$  do /* with  $RP = \langle a_1, \dots, a_n \rangle$  */
      if  $Prec(a_i) \subseteq S$  then
         $continue \leftarrow true$ ;
         $S \leftarrow S \uparrow a_i$ ;
         $plan \leftarrow plan \oplus \langle a_i \rangle$ 
      else
         $failed \leftarrow failed \oplus \langle a_i \rangle$ 
      endif
    endfor;
    if  $continue$  then
       $RP \leftarrow failed$ ;
       $failed \leftarrow \langle \rangle$ 
    else
       $RP \leftarrow \langle \rangle$ ;
      while  $\neg continue \wedge failed \neq \langle \rangle$  do
        forall  $f \in Add(First(failed))$  do
          if  $f \notin S \wedge \exists a \in (RP \oplus failed) \mid f \in Prec(a)$  then
            let  $actions =$ 
               $\{a \in O \mid f \in Add(a) \wedge Prec(a) \subseteq S\}$ ;
            if  $actions \neq \emptyset$  then
              let  $a = choose\_best(actions)$ ;
               $continue \leftarrow true$ ;
               $S \leftarrow S \uparrow a$ ;
               $plan \leftarrow plan \oplus \langle a \rangle$ ;
               $RP \leftarrow RP \oplus Rest(failed)$ ;
               $failed \leftarrow \langle \rangle$ 
            endif
          endif
        endfor;
        if  $\neg continue$  then
           $RP \leftarrow RP \oplus \langle First(failed) \rangle$ ;
           $failed \leftarrow Rest(failed)$ 
        endif
      endwhile
    endif
  endwhile
  return( $S, plan$ )
end

```

Figure 1: Lookahead algorithm

The second part is entered when no action has been applied in the most recent iteration of the first part. The goal is to try to repair the current (not applicable) relaxed plan, by replacing one action by another which is applicable in the current state  $S$ . Actions of  $failed$  are observed in turn, and we look for an action (in the global set of actions  $O$ ) applicable in  $S$ , which achieves an add effect of the action of  $failed$  we observe, this add effect being a precondition not satisfied in  $S$  of another action in the current relaxed plan. If several achievers are possible for the add effect of the action of  $failed$  we observe, we select the one that has the minimum cost in the relaxed planning graph used for extracting the initial relaxed plan (the cost of an action is the sum of the initial levels of its preconditions). When such an action is found, it is added to the lookahead plan and the global loop

is repeated. The action of  $failed$  observed when a repairing action was found is not kept in the current relaxed plan.

## Conclusion

We presented a new method for deriving information from relaxed plans, by the computation of lookahead plans. They are used in a complete best-first search algorithm for computing new nodes that can bring closer to a solution state. Although lookahead states are generally not goal states and the branching factor is increased with each created lookahead state, the experiments we conducted prove that in numerous domains from previous competitions (Rovers, Logistics, DriverLog, ZenoTravel, Satellite), our planner can solve problems that are up to ten times bigger (in number of actions of the initial state) than those solved by FF or by a classical best-first search without lookahead. YAHSP seems also to present good performances in domains from the 4<sup>th</sup> IPC, such as Pipesworld, Satellite and Promela/Philosophers where it solves all the problems, or Psr and Promela/Optical-Telegraph were a very few problems are not solved. The domain which seems to be the more difficult for YAHSP is Airport, where 12 problems are not solved yet. The counterpart for such improvements in performances and size of the problems that can be handled resides in the quality of solution plans that can be in some cases degraded (generally in domains where there are a lot of subgoal interactions). However, there are few of such plans and quality remains generally very good compared to FF.

## References

- Blum, A., and Furst, M. 1997. Fast planning through planning-graphs analysis. *Artificial Intelligence* 90(1-2):281–300.
- Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.
- Cayrol, M.; Régnier, P.; and Vidal, V. 2001. Least commitment in Graphplan. *Artificial Intelligence* 130(1):85–118.
- Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.
- Kautz, H., and Selman, B. 1999. Unifying SAT-based and Graph-based planning. In *Proc. IJCAI-99*, 318–325.
- Koehler, J.; Nebel, B.; Hoffmann, J.; and Dimopoulos, Y. 1997. Extending planning-graphs to an ADL subset. In *Proc. ECP-97*, 273–285.
- Long, D., and Fox, M. 1999. The efficient implementation of the plan-graph in STAN. *JAIR* 10:87–115.
- Nguyen, X.; Kambhampati, S.; and Nigenda, R. 2002. Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. *Artificial Intelligence* 135(1-2):73–123.
- Vidal, V. 2004. A Lookahead Strategy for Heuristic Search Planning. In *Proc. ICAPS-2004*.

# CPT: An Optimal Temporal POCL Planner based on Constraint Programming

**Vincent Vidal**

CRIL - Université d'Artois  
rue de l'université - SP16  
62307 Lens Cedex, FRANCE  
vidal@cril.univ-artois.fr

**Héctor Geffner**

ICREA & Universitat Pompeu Fabra  
Paseo de Circunvalacion 8  
08003 Barcelona, SPAIN  
hector.geffner@upf.edu

CPT is a new domain-independent temporal planner that combines a branching scheme based on Partial Order Causal Link (POCL) Planning with powerful and sound pruning rules implemented as constraints. Unlike other recent approaches that build on POCL planning (Nguyen & Kambhampati 2001; Younes & Simmons 2003), CPT is an optimal planner that minimizes makespan. The details of the planner and its underlying formulation are described in (Vidal & Geffner 2004) that is focused on the computation of 'canonical plans' where ground actions are not done more than once in the plan. The version used in the competition, removes this restriction and computes optimal temporal plans, whether canonical or not.

The development of CPT is motivated by the limitation of heuristic state approaches to parallel and temporal planning that suffer from a high branching factor (Haslum & Geffner 2001) and thus have difficulties matching the performance of planners built on SAT techniques such as Blackbox (Kautz & Selman 1999). In CPT, all branching decisions (resolution of open supports, support threats, and mutex threats), generate binary splits, and nodes  $\sigma$  in the search correspond to 'partial plans' very much as in POCL planning.

While ideally, one would like to have informative lower bounds  $f(\sigma)$  on the makespan  $f^*(\sigma)$  of the best complete plans that expand  $\sigma$ , so that the partial plan  $\sigma$  can be pruned if  $f(\sigma) \not\leq B$  for a given bound  $B$ , such lower bounds are not easy to come by in the POCL setting. CPT thus models the planning domain as a temporal constraint satisfaction problem, adds the constraint  $f^*(\sigma) \leq B$  for a suitable bound  $B$  on the makespan, and performs limited form of constraint propagation in every node  $\sigma$  of the search tree. The novelty of CPT in relation to other temporal POCL planners such as IxTET (Laborie & Ghalab 1995) and RAX (Jonsson *et al.* 2000), that also rely on constraint propagation (and Dynamic CSP approaches such as (Joslin & Pollack 1996)), is the formulation that enables CPT to reason about actions  $a$  that are not yet in the plan. Often a lot can be inferred about such actions including restrictions about their possible starting times and supports. Some of this information can actually be inferred

before any commitments are made; the lower bounds on the starting times of *all* actions as computed in Graphplan being one example (Blum & Furst 1995). CPT thus reasons with CSP variables that involve *all* the actions  $a$  in the domain and not only those present in the current plan, and for each such action, it deals with two variables  $S(p, a)$  and  $T(p, a)$  that stand for the possibly undetermined action supporting precondition  $p$  of  $a$ , and the possibly undetermined starting time of such an action. A causal link  $a'[p]a$  thus becomes a constraint  $S(p, a) = a'$ , which in turn implies that the supporter  $a'$  of precondition  $p$  of  $a$  starts at time  $T(p, a) = T(a')$ . A number of constraints enforce the correspondences among these variables. At the same time, the heuristic functions for estimating costs in a temporal setting, as introduced in (Haslum & Geffner 2001), are used to initialize variables domains and some 'distances' between actions (Van Beek & Chen 1999).

The CPT planner is implemented using the Choco CP library (Laburthe 2000) that operates on top of Claire, (Caseau, Josset, & Laburthe 1999), a high-level programming language that compiles into C++. Further details can be found in (Vidal & Geffner 2004) that is concerned mostly with the computation of optimal canonical plans; plans where no ground action is done more than once. The version of CPT used in the competition removes this restriction, and computes optimal temporal plans, whether canonical or not. Currently, the semantics of these plans follows the one in (Smith & Weld 1999) where interfering actions are not allowed to overlap in time. This condition has been relaxed in PDDL 2.1 where interfering actions may overlap sometimes (e.g., when preconditions do not have to be preserved throughout the execution of the action). We are currently trying to accommodate that semantics as well.

## References

- Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proceedings of IJCAI-95*, 1636–1642. Morgan Kaufmann.
- Caseau, Y.; Josset, F. X.; and Laburthe, F. 1999. Claire: Combining sets, search and rules to better express algo-

- rithms. In *Proceedings of the Int. Conf. on Logic Programming*.
- Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *Proc. European Conference of Planning (ECP-01)*, 121–132.
- Jonsson, A.; Morris, P.; Muscettola, N.; and Rajan, K. 2000. Planning in interplanetary space: Theory and practice. In *Proc. AIPS-2000*, 177–186.
- Joslin, D., and Pollack, M. E. 1996. Is "early commitment" in plan generation ever a good idea? In *Proceedings AAAI-96*, 1188–1193.
- Kautz, H., and Selman, B. 1999. Unifying SAT-based and Graph-based planning. In Dean, T., ed., *Proceedings IJCAI-99*, 318–327. Morgan Kaufmann.
- Laborie, P., and Ghallab, M. 1995. Planning with sharable resources constraints. In Mellish, C., ed., *Proc. IJCAI-95*, 1643–1649. Morgan Kaufmann.
- Laburthe, F. 2000. Choco: implementing a cp kernel. In *Proceedings CP-00, Lecture Notes in CS, Vol 1894*. Springer.
- Nguyen, X. L., and Kambhampati, S. 2001. Reviving partial order planning. In *Proc. IJCAI-01*.
- Smith, D., and Weld, D. 1999. Temporal planning with mutual exclusion reasoning. In *Proc. IJCAI-99*, 326–337.
- Van Beek, P., and Chen, X. 1999. CPlan: a constraint programming approach to planning. In *Proc. National Conference on Artificial Intelligence (AAAI-99)*, 585–590. AAAI Press/MIT Press.
- Vidal, V., and Geffner, H. 2004. Branching and pruning: An optimal temporal POCL planner based on constraint programming. In *Proceedings AAAI-04*. To appear.
- Younes, B. L. S., and Simmons, R. G. 2003. VHPOP: Versatile heuristic partial order planner. *Journal of AI Research* 20:405–430.

# BFHSP: A Breadth-First Heuristic Search Planner

Rong Zhou and Eric A. Hansen

Department of Computer Science and Engineering  
Mississippi State University  
Mississippi State, MS 39762  
{rzhou,hansen}@cse.msstate.edu

## Overview

Our Breadth-First Heuristic Search Planner (BFHSP) is a domain-independent STRIPS planner that finds sequential plans that are optimal with respect to the number of actions it takes to reach a goal. We developed BFHSP as part of our research on space-efficient graph search. It uses breadth-first search since we found that breadth-first search is more efficient than best-first search when divide-and-conquer solution reconstruction is used to reduce memory requirements. The specific search algorithm used by BFHSP is Breadth-First Iterative-Deepening A\* (Zhou & Hansen 2004) with some enhancements. Like HSP2.0 (Bonet & Geffner 2001a), BFHSP can search in either progression or regression space. The admissible heuristic function used is the  $h_{\max}$  heuristic (Bonet & Geffner 2001b) in progression search, and the *max-pair* heuristic (Haslum & Geffner 2000) in regression search.

## Divide-and-Conquer Solution Reconstruction

Our research objective in developing BFHSP is to design heuristic search algorithms that can find optimal plans using limited memory, especially in complex graphs with many duplicate paths where IDA\* is usually ineffective. BFHSP uses divide-and-conquer solution reconstruction to reduce its memory requirement. Divide-and-conquer solution reconstruction was first introduced to the heuristic search community by Korf (1999), based on a similar strategy used in dynamic programming algorithms for sequence comparison. The technique exploits the fact that it is not necessary to store all expanded nodes in a Closed list in order to prevent re-generation of already-expanded nodes. Instead, it suffices to store a subset of nodes that forms a *boundary* between the frontier and interior of the explicit search graph (Zhou & Hansen 2003).

Although nodes inside the boundary can be removed from memory without risking duplicate search effort, this means it is no longer possible to reconstruct a solution by the traditional traceback method. To allow divide-and-conquer solution reconstruction, each node stores information about a node along an optimal path to it that divides the problem in about half. Once the search problem is solved, information

about this midpoint node is used to divide the search problem into two subproblems: the problem of finding an optimal path from the start node to the midpoint node, and the problem of finding an optimal path from the midpoint node to the goal node. Each of these subproblems is solved by the same search algorithm, in order to find a node in the middle of their optimal path. The process continues recursively until primitive subproblems are reached, and all nodes on the optimal solution path have been identified. Since the time it takes to solve all subproblems is very short compared to the time it takes to solve the original search problem, this technique saves a great deal of memory in exchange for limited time overhead for solution reconstruction.

There are several different ways to store information about the midpoint node. BFHSP adopts the method used by Sparse-Memory A\* (Zhou & Hansen 2003). Each node stores a pointer to its predecessor or to an intermediate node along an optimal path, called a *relay node*, which is retained in memory. The advantage of this approach is that it takes less space and allows faster solution reconstruction.

## Breadth-First Heuristic Search

A significant difference between BFHSP and HSP2.0 is that BFHSP uses a breadth-first instead of the traditional best-first strategy of node expansion. This difference is based on our discovery that when divide-and-conquer solution reconstruction is used, breadth-first search is more memory-efficient than best-first search (Zhou & Hansen 2004). The reason for this is that memory requirements depend on the number of nodes needed to maintain a boundary between the frontier and interior of the search, and not the total number of nodes expanded. Figure 1 conveys an intuition of how breadth-first search results in a smaller set of boundary nodes. It shows that best-first node expansion “stretches out” the boundary, whereas breadth-first search does not and uses an upper bound to limit the width of the boundary. Although breadth-first search expands more nodes than best-first search, the memory it saves by storing a smaller boundary results in more efficient search.

Note that BFHSP uses both an admissible heuristic function and an upper bound to limit exploration of the search space. No node is inserted into the Open list if its  $f$ -cost is greater than an upper bound on the cost of an optimal solution, since such nodes cannot be on an optimal path.

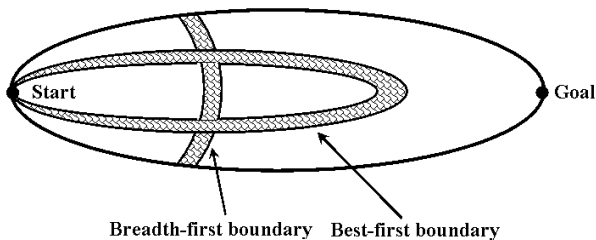


Figure 1: Comparison of best-first and breadth-first boundaries. The outer ellipse encloses all nodes with  $f$ -cost less than or equal to an (optimal) upper bound.

A breadth-first search graph divides into layers, one for each depth. To prevent duplicate search effort, BFHSP keeps (at least) three layers in memory: the currently-expanding layer, its immediate previous layer, and the next layer. In addition, it also stores a *relay layer* for the purpose of solution reconstruction. Other layers can be pruned to recover memory.

BFHSP provides two options regarding how previously-explored layers are removed from memory. The first option, called *aggressive pruning*, removes immediately any layer that is not one of the four layers mentioned previously. The second option, called *lazy pruning*, is the same as the first one, except that it removes layers only when memory is full. Because BFHSP with lazy pruning is the same as breadth-first branch-and-bound search until memory is full, the time overhead of solution reconstruction is avoided if there is enough memory. In IPC-4, BFHSP uses lazy pruning during solution reconstruction, because subproblems are often small enough in size that they can be solved by ordinary breadth-first branch-and-bound search.

For undirected graphs, storing only one previous layer is sufficient to prevent *all* duplicate search effort (Zhou & Hansen 2004). For directed graphs, the number of times a node can be re-generated by BFHSP is *at most* linear in the depth of the search. This contrasts sharply to the potentially exponential number of node re-regenerations for linear-space search algorithms that rely on depth-first search.

### Breadth-First Iterative-Deepening A\*

Although BFHSP uses an upper bound to limit its search space, it is possible to run the planner without a previously-computed upper bound. Instead, an iterative-deepening strategy can be used to avoid expanding nodes that have an  $f$ -cost greater than a hypothetical upper bound. *Breadth-First Iterative-Deepening A\** (BFIDA\*) first runs breadth-first heuristic search using the  $f$ -cost of the start node as an upper bound. If no solution is found, it increases the upper bound by one (or to the least  $f$ -cost of any unexpanded nodes) and repeats the search until a solution is found. In this respect, it is similar to Depth-First Iterative-Deepening A\* (Korf 1985). The difference is that it never expands the same node twice during the same iteration. (This claim holds for undirected graphs, and for many – but not all – directed graphs.) The amount of memory used is the same as the amount of memory BFHSP would use given an optimal

upper bound. However, BFIDA\* may run more slowly than BFHSP with a previously-computed upper bound, because running multiple iterations of BFHSP takes extra time.<sup>1</sup>

To reduce the number of iterations, BFHSP uses an improved version of BFIDA\*, called *BFIDA\*\_CR*, that is based on an idea used in IDA\*\_CR (Sarkar *et al.* 1991), where “CR” stands for controlled re-expansion. The idea is to create an algorithm in which the number of nodes expanded in successive iterations increases exponentially with the number of iterations. Among other things, BFIDA\*\_CR has an interesting advantage over IDA\*\_CR. That is, for planning problems with unit action cost, BFIDA\*\_CR can guarantee that the first solution found is optimal, because it uses breadth-first search; whereas IDA\*\_CR cannot, due to its use of depth-first search.

Unlike conventional iterative-deepening search, which increases its bound to the minimum  $f$ -cost of any unexpanded nodes after each iteration, BFIDA\*\_CR may use a slightly higher bound to reduce overall node expansions by reducing the number of iterations it takes to find a solution. The benefit of using BFIDA\*\_CR is most evident in problems with small branching factor but long solution depth, such as the newly-released *airport* domain in IPC-4.

### Admissible Search Heuristics

BFHSP uses the admissible  $h_{\max}$  heuristic (Bonet & Geffner 2001b) in progression search and the *max-pair* heuristic (Haslum & Geffner 2000) in regression search. In addition, we implemented the *max-triple* heuristic for regression search by considering triples (instead of pairs) of atoms. The max-triple heuristic is more accurate than the max-pair heuristic, and often results in four or five-fold reduction in node expansions. The max-triple heuristic is, however, more time-consuming to compute and takes more memory to store, because its time (and space) complexity is cubic in the number of atoms. As a result, it is not the default search heuristic in BFHSP. An interesting observation, however, is that using the max-triple heuristic lets BFHSP solve some STRIPS instances of the *philosophers* problem that cannot be solved by using the max-pair heuristic in regression search, because using the max-triple heuristic makes it possible to recognize high-order mutexes (Blum & Furst 1995) and to prune states that contain them.

### Special Features

Breadth-first (heuristic) search, when applied to problems with unit action cost, has the advantage that when a node is first generated, an optimal path to it has been found. With some changes to the algorithm, this property can be exploited to reduce the internal memory requirement of BFHSP. In fact, we have developed an external-memory version of BFHSP that uses disk storage in order to bound its internal-memory requirement (Forthcoming). However, we did not use it in IPC-4, because given the constraints of the Competition (30 minutes of CPU time and 1 gigabytes of

<sup>1</sup>It is possible to improve the efficiency of BFHSP by reusing information stored from previous iterations of BFIDA\*, but we did not explore this possibility in our current implementation.



RAM), it is unclear whether memory is the bottleneck instead of time. In our experience with IPC-4, there are more problems for which BFHSP ran out of time before it ran out of memory, than the other way around.

## Conclusion

Our primary design goal for BFHSP is to reduce its memory requirement, which is an important issue for many optimal heuristic search-based planners. Unfortunately, the time and space constraints of this Competition do not make it possible to fully demonstrate the advantages of BFHSP. For example, we have run BFHSP for days without running out of memory and have used it to find optimal plans for STRIPS problems that are far beyond the reach of HSP2.0 or HSP<sup>+</sup> (Haslum & Geffner 2000). We believe that in many real-world applications where optimality is important, memory is likely to be a bottleneck, and BFHSP will have an advantage over other optimal planners.

## Acknowledgement

We thank Blai Bonet and Hector Geffner for making publicly available their code for HSP2.0, upon which BFHSP is built.

## References

- Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proc. of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1636–42.
- Bonet, B., and Geffner, H. 2001a. Heuristic search planner 2.0. *AI Magazine* 22(3):77–80.
- Bonet, B., and Geffner, H. 2001b. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.
- Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. of the 5th International Conference on AI Planning and Scheduling*, 140–149.
- Korf, R. 1985. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence* 27:97–109.
- Korf, R. 1999. Divide-and-conquer bidirectional search: First results. In *Proc. of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1184–1189.
- Sarkar, U.; Chakrabarti, P.; Ghose, S.; and Sarkar, S. D. 1991. Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence* 50:207–221.
- Zhou, R., and Hansen, E. 2003. Sparse-memory graph search. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, 1259–1266.
- Zhou, R., and Hansen, E. 2004. Breadth-first heuristic search. In *Proc. of the 14th International Conf. on Automated Planning and Scheduling*.

# Heuristic Planning via Roadmap Deduction

Lin Zhu and Robert Givan \*

Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907 USA  
{lzhu, givan}@purdue.edu

## Abstract

Porteous et al. (2001) introduced the concept of “planning landmarks”—propositions that must be true at some point during the execution of every successful plan. We define “relaxed landmarks,” a subset of the planning landmarks, and give a sound and complete algorithm for computing relaxed landmarks. All the landmarks computed by the previous method are relaxed landmarks, but that method was significantly incomplete for finding relaxed landmarks. We additionally discriminate between useful “causal” landmarks and misleading “non-causal” landmarks, and our method easily omits the latter. We then present a novel method for partially ordering landmarks into “landmark roadmaps”, where two ordered landmarks are present in the given order in every successful plan execution. Finally, we give an efficient means of extending FF’s heuristic to leverage a landmark roadmap by weighting the components of the relaxed plan. The SCHEME variant of FF using this heuristic, ROADMAPPER, works on the non-temporal ADL versions of the IPC4.

Our ROADMAPPER planner is a variant of FF where the heuristic is significantly more complex and derived from a partially ordered set of landmarks. In what follows, we formalize, motivate, and define the heuristic used in ROADMAPPER.

## Background

We refer to (McAllester & Rosenblitt 1991) as SNLP and generally follow and adapt it for notation regarding STRIPS planning and partial order planning.

**Strips Planning.** Let  $X$  be a finite set of propositions. A state  $S$  is a finite subset of  $X$ . An action  $o$  is a triple  $o = \langle \text{PRE}(o), \text{ADD}(o), \text{DEL}(o) \rangle$  where  $\text{PRE}(o)$  are the *preconditions*,  $\text{ADD}(o)$  is the *add list* and  $\text{DEL}(o)$  is the *delete list*, each being a set of propositions. The result  $\text{RESULT}(S, (o_1, \dots, o_n))$  of applying an action sequence  $(o_1, \dots, o_n)$  to a state  $S$  is given by  $\text{RESULT}(\text{RESULT}(S, (o_1, \dots, o_{n-1})), (o_n))$ , where for  $n$  equals 1 the result is undefined unless  $\text{PRE}(o_1) \subseteq S$ , and  $(S \cup \text{ADD}(o_1)) - \text{DEL}(o_1)$ , otherwise.

\*We are grateful to Alan Fern and Matthew Greig for useful discussions.

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

A *planning task*  $P$  is a set of actions containing actions START and FINISH, where  $\text{PRE}(\text{START})$ ,  $\text{DEL}(\text{FINISH})$ , and  $\text{ADD}(\text{FINISH})$  are all empty. We refer to  $\text{PRE}(\text{FINISH})$  as the *goal region* and  $\text{ADD}(\text{START})$  as the *initial state*. We also consider the *relaxed planning task*  $P^R$  (which ignores delete effects) given by  $\{(\text{PRE}(o), \text{ADD}(o), \emptyset) \mid (\text{PRE}(o), \text{ADD}(o), \text{DEL}(o)) \in P\}$ .

A *linear solution* for a task  $P$  is an ordered action sequence  $\vec{o}$ , beginning with START, ending with FINISH such that  $\text{RESULT}(\emptyset, \vec{o})$  is defined.

**Partial Order Planning** To allow multiple occurrences of the same action or the same proposition within our nonlinear plans, we introduce finite sets of *step names* and *fact names*, respectively. Each plan includes a *symbol table* mapping step names to actions and fact names to propositions<sup>1</sup>. We use step names and fact names as actions or propositions, respectively, assuming an implicit look-up of the corresponding action or proposition in the appropriate symbol table. Note that naming for facts is needed so that we can later allow a fact to be a landmark more than one time, indicating that that fact must be added multiple times in any successful plan.

A *nonlinear plan*, or *plan* for short, is a pair  $\langle \Sigma, \leq \rangle$  of a symbol table  $\Sigma$ , and a partial order<sup>2</sup>  $\leq$  on names<sup>3</sup> (step names and fact names) in  $\Sigma$ . We write  $x < y$  to abbreviate  $x \leq y \wedge x \neq y$ . The *length* of the plan is the number of step symbols in  $\Sigma$ .

SNLP introduced the concept of *causal links* to help developing a systematic, sound and complete search algorithm. Causal links can be inferred from our representation. A *causal link* is a triple  $\langle s, p, w \rangle$ , written as  $s \xrightarrow{p} w$ , where  $s$  and  $w$  are step names, and  $p$  is a proposition<sup>4</sup> in  $\text{ADD}(s) \cap \text{PRE}(w)$ , such that  $s < x < w$  for some  $x$  mapped to  $p$ , and that either  $v < s$  or  $w < v$  for every step name  $v$  in the set  $\{y \in \Sigma - X \mid p \in \text{DEL}(y)\} - \{s, w\}$ . Note, there can be two different causal links  $s_1 \xrightarrow{p} w$  and  $s_2 \xrightarrow{p} w$  for

<sup>1</sup>This is different from the original SNLP paper, where the symbol table contained only step names.

<sup>2</sup>For our purpose, a partial order is a reflexive, transitive, anti-symmetric relation, viewed as a set of orders  $x < y$ .

<sup>3</sup>Again, ordering on fact names is necessary to allow proposition landmarks.

<sup>4</sup>Note, not a fact name.

the same step name  $w$  and the same proposition  $p$ . This is not the case for SNLP.

A bijection  $\sigma$  on names is called a *renaming*. We extend such renamings naturally to bijections on complex objects containing names (such as plans), in each case renaming the names appearing within. We say a nonlinear plan  $\langle \Sigma', \leq' \rangle$  *refines*  $\langle \Sigma, \leq \rangle$  whenever, for some renaming  $\sigma$ ,  $\sigma(\Sigma) \subseteq \Sigma'$  and  $\sigma(\leq) \subseteq \leq'$ . If either of the containment is proper, the refinement is called *strict*.

A nonlinear plan is called *complete* if FINISH is named by  $\Sigma$ , and for every step name  $v \in \Sigma$  and every proposition  $p$  in  $\text{PRE}(v)$ , there is *at least one* causal link  $s \xrightarrow{p} v$ . Later in this paper, we generally restrict our attention to nonlinear plans that are complete.

A *relaxed (nonlinear) plan* for  $P$  is a nonlinear plan for the corresponding relaxed task  $P^R$ . Obviously every plan for  $P$  is a relaxed plan for  $P$ . A relaxed plan is called *non-redundant* if any proposition or action is named at most once. Any relaxed plan refines some non-redundant relaxed plan.

## Landmarks and Roadmaps

**Definition 1** A nonlinear plan  $\langle \Sigma, \leq \rangle$  is a roadmap for planning task  $P$  if every complete nonlinear plan for  $P$  refines  $\langle \Sigma, \leq \rangle$ .

We call actions or propositions appearing in  $\Sigma$  for a roadmap *causal landmarks*. Causal landmarks that are propositions are *landmarks* in the sense of Porteous et al. 2001: the planning problem cannot be solved if the actions adding such a proposition are removed. However, not every landmark is a causal landmark: some landmarks are just “incidental” effects of the action that adds them. Consider a problem where the agent must travel in the rain to solve the problem. “Getting wet” will be a non-causal landmark, as it is a necessary effect of an essential action. Setting “getting wet” as a subgoal would be misleading. Thus we consider non-causal landmarks to be misleading and inappropriate as subgoals for the planning task.

Porteous et al. showed the problem of finding landmarks for a planning task to be PSPACE-hard. The proof can be easily extended here.

**Theorem 1** The problem of deciding whether a proposition or an action is a causal landmark is PSPACE-hard.

Therefore deducing any nontrivial roadmap is difficult as well. Here we will concentrate on a tractable subset of roadmaps.

**Definition 2** A relaxed roadmap for planning task  $P$  is a roadmap for the corresponding relaxed planning task  $P^R$ .

We call actions or propositions appearing in  $\Sigma$  for a relaxed roadmap *relaxed causal landmarks*. Every relaxed roadmap is a roadmap, and therefore every relaxed causal landmark is a causal landmark.

To compute relaxed roadmaps, we first assume a base algorithm  $\text{A\_RELAXED\_PLAN}(P)$  that finds some *non-redundant* plan for the relaxed planning task  $P^R$ , if there exists one, and returns FALSE otherwise. The heuristic

computation in FF contains an efficient implementation of  $\text{A\_RELAXED\_PLAN}$ , which empirically often returns a good approximation of the shortest relaxed plan.

The relaxed roadmap is computed in a generate-and-test way. We first call  $\text{A\_RELAXED\_PLAN}$  to generate a relaxed plan  $\langle \Sigma, \leq \rangle$ . Since by definition any relaxed roadmap is refined by  $\langle \Sigma, \leq \rangle$ , we select a subset of  $\Sigma$  and a subset of  $\leq$  to get a relaxed roadmap, by the test phase described below.

Again, the function  $\text{A\_RELAXED\_PLAN}$  is used to test whether a proposition or an action is a relaxed causal landmark. To do so, we first define the reduced planning problem  $P_{\bar{x}}$ , intended to be solvable exactly when  $x$  is *not* a causal landmark for  $P$ . If the landmark  $x$  which we want to test is a proposition,  $P_{\bar{x}}$  is  $\{o_{\bar{x}} = \langle \text{PRE}(o), \text{ADD}(o) - \{x\}, \text{DEL}(o) \rangle \mid o \in P\}$ ; otherwise  $P_{\bar{x}} = P - \{x\}$ . We know  $x$  is a relaxed causal landmark for  $P$  if and only if  $\text{A\_RELAXED\_PLAN}(P_{\bar{x}})$  returns FALSE.<sup>5</sup>

Further, we can use the above method to verify  $x < y$  for a (relaxed) roadmap. To do so, we define  $P_{\rightarrow y}$ , the subproblem of  $P$  with goal of reaching  $y$ .  $P_{\rightarrow y}$  is the same as  $P$  except that FINISH is replaced with  $\langle \text{PRE}(y), \emptyset, \emptyset \rangle$  if  $y$  is a step name, and  $\langle \{y\}, \emptyset, \emptyset \rangle$  otherwise. For every pair of causal landmarks  $x$  and  $y$ , we know that  $x < y$  appears in a relaxed roadmap if and only if  $x$  is a relaxed causal landmark of  $P_{\rightarrow y}$ , i.e.,  $\text{A\_RELAXED\_PLAN}(P_{\bar{x}, \rightarrow y})$  returns FALSE.

In the algorithm below, we use  $R^*$  to denote the reflexive transitive closure of a relation  $R$ .

**Algorithm 1**  $\text{RELAXED\_ROADMAP}(P)$

*Input: A planning task*

```

 $\langle \Sigma_c, \leq_c \rangle \leftarrow \text{A\_RELAXED\_PLAN}(P)$ 
 $\Sigma_r \leftarrow \{x \in \Sigma_c \mid$ 
    not  $\text{A\_RELAXED\_PLAN}(P_{\bar{x}})\}$ 
 $\leq_r \leftarrow \{(x, y) \in \leq_c \cap \Sigma_r \times \Sigma_r \mid$ 
    not  $\text{A\_RELAXED\_PLAN}(P_{\bar{x}, \rightarrow y})\}$ 
return  $\langle \Sigma_r, \leq_r^* \rangle$ 

```

The bound on the times of calling  $\text{A\_RELAXED\_PLAN}(P)$  is  $O(n + m^2)$ , where  $n$  is the total number of actions and propositions, and  $m$  is the total number of relaxed landmarks. In practice,  $m$  is typically much smaller than  $n$ . There are several ways to make this computation more efficient which are omitted here.

**Theorem 2** The output of  $\text{RELAXED\_ROADMAP}(P)$

*Soundness* is a relaxed roadmap for  $P$ , and

*Completeness* refines every relaxed roadmap for  $P$ .

We note here that the above method is not the only to deduce roadmaps. Roadmaps generated in other ways can be incorporated.

## Weighted Relaxed Plan Length as a Heuristic

A roadmap intuitively contains important ordered subgoals of a planning problem. Porteous et al. 2001 proposed to use it to sub-divide planning problems into smaller, easier

<sup>5</sup>In contrast, Porteous et al. define  $P_{\bar{x}}$  as  $P - \{o \mid x \in \text{ADD}(o)\}$  if  $x$  is a proposition. This can be used to test landmarks, but cannot distinguish causal landmarks.

pieces, and then use a base planner to solve them one by one. This methodology, however, ignores the interactions between solving subgoals. In particular, the base planner may solve a subgoal in a way so that later subgoals become hard or impossible to solve.

Another way to utilize landmarks is to simply use the number of landmarks as a heuristic guiding forward search. Empirical results show it is effective on some domains at a high level (Zhu & Givan 2003). However, this heuristic is not informative on how to solve the subgoals. It is only when a subgoal is solved, by blind search, that the heuristic decreases by one.

We introduce a novel usage of roadmaps below. We use roadmaps to weight the components of a successful heuristic, emphasizing solving one subgoal, while keeping an eye on the solution of other subgoals.

The success of FF (Hoffmann & Nebel 2001) mainly comes from its efficient and accurate heuristic, and its unique search strategy, *enforced hill-climbing*, that is incomplete but often very fast<sup>6</sup>. Unlike pure hill-climbing, which iteratively selects single actions with the best one-step-look-ahead heuristic value and often has difficulty with local minima and plateaus, enforced hill-climbing iteratively uses breadth-first search to find *action sequences* that lead to states with heuristic values that are strictly better than the current state.

Here, we discuss FF's heuristic and our way to improve its quality. We know that an ideal search heuristic would be the optimal length of a complete plan. Since this heuristic is not tractably computable, FF approximates it by two relaxations. In the following discussion, we denote the set of plans for task  $P$  by  $PLANS(P)$ , and the set of relaxed plans by  $RELAXED\_PLANS(P)$ . Obviously  $PLANS(P) \subseteq RELAXED\_PLANS(P)$ .

First, FF considers the relatively easier problem of computing  $RELAXED\_PLANS(P)$ , and approximates (and lower bounds) the optimal length among  $PLANS(P)$  by the optimal length among  $RELAXED\_PLANS(P)$ . Empirical (Hoffmann 2001) and theoretical (Hoffmann 2002) results show that optimal relaxed plan length (applied with enforced hill-climbing) is a good heuristic for a large variety of planning domains, and often leads to polynomial search complexity.

Second, since it's still difficult to compute the optimal relaxed plan, it extracts one relaxed plan to get an approximation of the optimal relaxed plan length, utilizing various heuristic considerations to encourage near-optimality. Empirical results (Hoffmann 2001) show that the length of the relaxed plan extracted this way is often a good approximation of the optimal relaxed plan length. FF uses this length as its heuristic.

We extend the relaxed-plan-length heuristic by assigning weights to its components. Among all the landmarks that have no other landmark ordered before them in the roadmap, we choose one achievable by the shortest relaxed plan. The heuristic of the global problem is the weighted sum of relaxed plan lengths of all landmarks. The chosen landmark

gets weight  $f$ , and all the others get weight 1. We generally consider  $f$  that is greater than 1. The greater  $f$  is, the more aggressive the planner is on solving one subgoal, and the more oblivious it is to the difficulty of other subgoals.

In theory and in practice, the computation of this heuristic should add only trivial burden to that of FF, besides the one-time cost of computing roadmap.

We then utilize this heuristic in a similar way to FF, and apply the resulting planner, ROADMAPPER, to non-temporal ADL versions of the fourth international planning competition. Our implementation is fully written in SCHEME, a dialect of LISP.

## References

- Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.
- Hoffmann, J. 2001. Local search topology in planning benchmarks: An empirical analysis. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, 453–458.
- Hoffmann, J. 2002. Local search topology in planning benchmarks: A theoretical analysis. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02)*. 379–387.
- McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, volume 2, 634–639. Anaheim, California, USA: AAAI Press/MIT Press.
- Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In *Recent Advances in AI Planning. 6th European Conference on Planning (ECP'01)*, 37–48.
- Zhu, L., and Givan, R. 2003. Landmark Extraction via Planning Graph Propagation. In *Printed Notes of ICAPS'03 Doctoral Consortium*. Trento, Italy.

<sup>6</sup>In the rare case the *enforced hill-climbing* fails, FF resorts to an expensive but complete search.

# Probabilistic Part

# Introduction to the Probabilistic Planning Track

**Michael L. Littman**

Department of Computer Science  
Rutgers University  
Piscataway, NJ 08854 USA  
mlittman@cs.rutgers.edu

**Håkan L. S. Younes**

Computer Science Department  
Carnegie Mellon University  
Pittsburgh, PA 15213 USA  
lorens@cs.cmu.edu

## Abstract

The 2004 International Planning Competition, IPC-4, includes a probabilistic planning track for the first time. We briefly summarize the design of the track.

## Introduction

Domain-independent planners seek to synthesize plans that achieve goals as cheaply as possible. While classical planning is concerned with domains in which operators have deterministic effects—the planner can predict with certainty how its decisions will change the environment—work on probabilistic planning expands the field to include operators with uncertain effects. The inclusion of probabilistic effects extends domain description languages to a more realistic class of applications. However, this increased generality comes with the price of increased computational complexity of planners and plan evaluation (Littman, Goldsmith, & Mundhenk 1998).

The 2004 International Planning Competition, IPC-4, introduces a probabilistic planning track for the first time. The goal of the track is to provide a forum for the evaluation and comparison of approaches to probabilistic planning. At the time of this writing, most of the logistical decisions have been made, but the competition and evaluation have not yet taken place. This document summarizes the status of the competition as of April 2004. For the latest developments, please visit: <http://www.cs.rutgers.edu/~mlittman/topics/ipc04-pt/>.

The probabilistic track was organized by the authors, Michael L. Littman and Håkan L. S. Younes, and a team at Rutgers consisting of John Asmuth, David Weissman, and Paul Batchis.

## Calendar

Planning for the probabilistic track dates back to shortly after IPC-1. However, it was Sven Koenig and Shlomo Zilberstein's idea to specifically create a probabilistic track for IPC-4. Initial attempts to drum up support for the competition in 2002 led to the creation of a mailing list with addresses of 87 interested researchers. As the form of the competition itself took shape, potential participants were asked to register in September 2003. Representatives from

22 groups (spread over 4 continents) signed up to receive the first version of the PPDDL validation software.

In April 2004, we held a “mock competition” as a way of identifying the most committed groups and for testing our evaluation procedure. Six groups participated (groups C (UMass), E (Dresden), G (ANU), J (Purdue), P (Simon Bolivar) and, D (Bowdoin)). Several other groups expressed regrets that their planners were not yet ready. As of this writing, several groups have explicitly pulled out of the competition and 15 groups remain signed up. We're expecting between 5 and 10 groups to participate in the competition within the next three weeks.

## Domain Description Language

We intended the competition to be accessible to researchers studying “factored” or first-order Markov decision processes (extensions of MDPs to predicate-based state representations) and decision-theoretic planning (extensions of classical planning to uncertain effects and utilities). The state of the art in evaluating classical planners is the IPC and their choice of domain description languages is PDDL (Fox & Long 2001). We sought to introduce a minimal set of extensions to PDDL2.1 to support probabilistic effects. The probabilistic planning domain description language (PPDDL1.0) we developed is described in the following paper.

PPDDL1.0 extends PDDL2.1 to support the succinct representation of Markov decision processes. However, for this first competition, we decided to restrict the set of language features that participants would need to support. Specifically, the evaluation domains included neither numeric state variables nor hidden propositions. As such, there is a direct conversion from the provided PPDDL specifications to finite (though perhaps enormous) MDPs.

To support the programming efforts of the participants, we provided C++ code for parsing PPDDL domains and problems and an mtbdd-based converter from PPDDL to a propositionalized MDP representation. We believe several participants wrote their own parsers and converters and others used our initial code to varying degrees.

## Objectives

Each domain used in the competition came in one of two possible styles. In *goal-only* domains, a goal specification

was provided and the objective of the planner was to reach a goal state. Planners in these domains are evaluated by estimating the probability that they will reach a goal state. Such domains can be viewed as a type of MDP in which a unit reward value is provided upon arrival in a goal state and all other transitions result in zero reward.

The second, and more common, style of domain in the competition was “reward goal” problems. These domains include operators with state-independent cost, a goal specification, and a goal-reward value issued upon arrival in a goal state. Although PPDDL supports positive and negative state-dependent rewards as well as continuing tasks with no terminating goal state, we thought restricting objectives as described kept them as close as possible in spirit to the kinds of objectives supported in the classical track.<sup>1</sup> By assigning goal rewards, each execution of a planner on a problem terminates with a total reward value, with early termination preferred to longer execution traces. Planners are compared according to their total expected reward, computed as the sum of the goal reward (if obtained) minus any action costs.

We also planned to support evaluation of “nondeterministic” domains. However, as no groups stepped up to participate in such a track, we did not pursue it.

## Evaluation

In classical planning, a plan is a series of operators. A valid plan is one that, when applied to the initial state, achieves the goal. Because of the uncertainty in state transitions, straight-line plans are often not appropriate in probabilistic domains. Although several groups have expressed an intention to synthesize only unconditional plans, we did not want to impose any particular plan representation on participants.

We decided to evaluate planners by sampling or simulation. That is, our plan validator is a server, and individual planning/execution algorithms connect to the evaluator as clients. They initiate a session by providing an agreed upon domain id, receive an initial state, and return an operator. The server-client dialog continues until a terminating condition is reached at which point the validator evaluates the performance of the planner. This entire process is repeated several times with results averaged over the multiple runs.

Source code for a server (“mdpsim”) was provided to all participants and updated as changes were made to the domain description language and evaluation procedure. For official evaluations runs, a server was run at Rutgers with participants connecting via the Internet. In trial runs, participants reported communication times ranging from 20ms (CMU) to 76ms (South America) to 230ms (Australia) roundtrip. To compensate for the wide range of communication times, participants were offered the option of temporary accounts at CMU to install and run their clients.

Based on feedback from the mock competition, we decided to evaluate each planner in each domain in a 15-minute block. During this block, planners can carry out any computation, pre-processing, or plan generation that they choose to do. However, they must also execute 30 runs from an initial state to a goal state (voluntary premature termination is also

an option). The average reward obtained over these 30 runs (with zero reward for any runs that were not taken) is the planner’s evaluation score.

We chose 30 runs because this number may provide sufficient statistical confidence to distinguish between planners. We did not subdivide the 15 minutes into 30-second blocks to allow participants to amortize planning effort over multiple runs. We suspect that most planners will use the majority of the 15 minutes to construct a plan and the remainder to evaluate the plan 30 times. However, the evaluation procedure supports a wide variety of strategies.

## Domains

In the mock competition, we included 19 test problems: blocksworld (5 5-block problems, 5 25-block problems, and 5 125-block problems), one colored blocksworld problem, one fileworld problem, a variation of the coffee domain (Dearden & Boutilier 1997), and a variation of the sandcastle problem (Majercik & Littman 1998). These include problems with and without functions and both goal-only and reward-goal domains.

The blocksworld problems were created using a blocksworld problem generator that we developed. It will be available after the conference on the competition website. We have also released a logistics domain generator we call “boxworld”. Problems generated from the blocksworld and boxworld generators will be included in the competition. Because these generators were released in advance, participants have the option of learning or hand-tuning rules for their planners to exploit structure in these domains.

Several other domains will be included in the competition, to be distributed immediately prior to evaluation. All domains we used for evaluation will be made available to interested researchers. Visit our web site or contact us by email for more information.

## Acknowledgements

This work was supported in part by NSF grants IIS-0329153 and IIS-0315909. We thank the ICAPS and IPC organizers for their support and encouragement and the participants for their enthusiasm and creativity.

## References

- Dearden, R., and Boutilier, C. 1997. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence* 89(1–2):219–283.
- Fox, M., and Long, D. 2001. PDDL2.1: An extension to PDDL for expressing temporal planning domains. Technical report, University of Durham, UK.
- Littman, M. L.; Goldsmith, J.; and Mundhenk, M. 1998. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research* 9:1–36.
- Majercik, S. M., and Littman, M. L. 1998. MAXPLAN: A new approach to probabilistic planning. In Simmons, R.; Veloso, M.; and Smith, S., eds., *Proceedings of the Fourth International Conference on Artificial Intelligence Planning*, 86–93. AAAI Press.

<sup>1</sup>Thanks to Héctor Geffner for sharing this observation.

# PPDDL1.0: The Language for the Probabilistic Part of IPC-4

**Håkan L. S. Younes**

Computer Science Department  
Carnegie Mellon University  
Pittsburgh, PA 15213, USA  
lorens@cs.cmu.edu

**Michael L. Littman**

Department of Computer Science  
Rutgers University  
Piscataway, NJ 08854, USA  
mlittman@cs.rutgers.edu

## Introduction

A standard domain description language, PDDL (Ghallab *et al.* 1998; McDermott 2000; Fox & Long 2003), for deterministic planning domains has simplified sharing of domain models and problems in the planning community, and has enabled direct comparisons of different planning systems. As a result, there has been considerable progress in planning research with deterministic domain models since the first International Planning Competition in 1998.

The 4th International Planning Competition includes a probabilistic track for the first time in an attempt to create a common platform for the evaluation of probabilistic and decision-theoretic planning systems. This document briefly describes the input language, PPDDL1.0, that was used for the probabilistic track. PPDDL1.0 is essentially a syntactic extension of levels 1 and 2 of PDDL2.1 (Fox & Long 2003). We assume that the reader is familiar with PDDL2.1, so focus on the new language features, which include probabilistic effects and rewards. The semantics of a PPDDL1.0 planning problem is given in terms of a Markov decision process (Howard 1960).

Note that, unlike PDDL2.1, we do not impose a specific structure on plans in PPDDL1.0. Planning systems are evaluated using a client-server model in the probabilistic track of the competition. During evaluation of a planner, the server send a state to the client (planning system), which in return sends an action to be executed in the given state. The problem of plan representation is left entirely to the planning systems.

## Probabilistic Effects

In order to define probabilistic and decision-theoretic planning problems, we need to add support for probabilistic effects. The syntax for probabilistic effects is

`(probabilistic  $p_1$   $e_1$  ...  $p_k$   $e_k$ )`

meaning that effect  $e_i$  occurs with probability  $p_i$ . We require that the constraints  $p_i \geq 0$  and  $\sum_{i=1}^k p_i = 1$  are fulfilled: a probabilistic effect declares an exhaustive set of probability-weighted outcomes. However, we allow a probability-effect pair to be left out if the effect is empty. In other words,

`(probabilistic  $p_1$   $e_1$  ...  $p_l$   $e_l$ )`

with  $\sum_{i=1}^l p_i \leq 1$  is syntactic sugar for

Name	Type	Init 1	Init 2
<i>bomb-in-package</i> <sub>package1</sub>	boolean	true	false
<i>bomb-in-package</i> <sub>package2</sub>	boolean	false	true
<i>toilet-clogged</i>	boolean	false	false
<i>bomb-defused</i>	boolean	false	false

Table 1: State variables and their initial values for the “Bomb and Toilet” problem.

`(probabilistic  $p_1$   $e_1$  ...  $p_l$   $e_l$   $q$  (and))`

with  $q = 1 - \sum_{i=1}^l p_i$ . For example, the effect

`(probabilistic 0.9 (clogged))`

means that with probability 0.9 the state variable *clogged* becomes true in the next state, while with probability 0.1 the state remains unchanged. Outcomes are not required to be mutually exclusive. A new requirements flag is introduced to signal that support for probabilistic effects is required:

`:probabilistic-effects`

Figure 1 shows an encoding in PPDDL of the “Bomb and Toilet” example described by Kushmerick, Hanks, & Weld (1995). In this problem, there are two packages, one of which contains a bomb. The bomb can be defused by dunking the package containing the bomb in the toilet. There is a 0.05 probability of the toilet becoming clogged when a package is placed in it. The problem definition in Figure 1 also shows that initial conditions in PPDDL can be probabilistic. In this particular example we define two possible initial states with equal probability (0.5) of being the true initial state. Table 1 lists the state variables for the “Bomb and Toilet” problem and their values in the two possible initial states. Intuitively, we can think of the initial conditions of a PPDDL planning problem as being the effects of an action forced to be scheduled right before time 0. Also, note that the goal of the problem involves negation, which is why the problem definition declares the `:negative-preconditions` requirements flag.

PPDDL allows arbitrary nesting of conditional and probabilistic effects. This is in contrast to popular propositional encodings, such as probabilistic STRIPS operators (PSOs) (Kushmerick, Hanks, & Weld 1995) and factored PSOs (Dearden & Boutilier 1997), which do not allow conditional effects nested inside probabilistic effects. While arbitrary



```

(define (domain bomb-and-toilet)
  (:requirements :conditional-effects :probabilistic-effects)
  (:predicates (bomb-in-package ?pkg) (toilet-clogged) (bomb-defused))
  (:action dunk-package
    :parameters (?pkg)
    :effect (and (when (bomb-in-package ?pkg) (bomb-defused))
      (probabilistic 0.05 (toilet-clogged)))))

(define (problem bomb-and-toilet)
  (:domain bomb-and-toilet)
  (:requirements :negative-preconditions)
  (:objects package1 package2)
  (:init (probabilistic 0.5 (bomb-in-package package1)
    0.5 (bomb-in-package package2)))
  (:goal (and (bomb-defused) (not (toilet-clogged)))))

```

Figure 1: PPDDL encoding of “Bomb and Toilet” example.

nesting does not add to the expressiveness of the language, it can allow for exponentially more compact representations of certain effects given the same set of state variables and actions (Rintanen 2003). However, any PPDDL action can be translated into a set of PSOs with at most a polynomial increase in size of the representation. Consequently, it follows from the results of Littman (1997) that PPDDL is representationally equivalent to dynamic Bayesian networks (Dean & Kanazawa 1989), which is another popular representation for MDP planning problems.

## Rewards and Plan Objectives

Markovian rewards, associated with state transitions, can be encoded using fluents. PPDDL reserves the fluent *reward*, accessed as `(reward)` or *reward*, to represent the total accumulated reward since the start of execution. Rewards are associated with state transitions through update rules in action effects. The use of the *reward* fluent is restricted to action effects of the form

( *<additive-op>* *<reward fluent>* *<f-exp>* )

where *<additive-op>* is either *increase* or *decrease*, and *<f-exp>* is a numeric expression not involving *reward*. Action preconditions and effect conditions are not allowed to refer to the *reward* fluent, which means that the accumulated reward does not have to be considered part of the state space. The initial value of *reward* is zero. These restrictions on the use of the *reward* fluent allow a planner to handle domains with rewards, without having to implement full support for fluents.

The requirements flag, `:rewards`, is introduced to signal that support for Markovian rewards is required. Domains that require both probabilistic effects and rewards can declare the `:mdp` requirements flag, which implies `:probabilistic-effects` and `:rewards`.

Figure 2 shows part of the PPDDL encoding of a coffee delivery domain described by Dearden & Boutilier (1997). A reward of 0.8 is awarded if the user has coffee when the “buy-coffee” action is executed, and a reward of 0.2 is awarded when “buy-coffee” is executed in a state where

*is-wet* is false. Note that a total reward of 1.0 can be awarded as a result of executing the “buy-coffee” action if it is executed in a state where both *user-has-coffee* and *¬is-wet* hold.

Action effects with inconsistent transition rewards are not permitted. For example, the effect `(probabilistic 0.5 (increase (reward) 1))` is semantically invalid because it associates a reward of both 1 and 0 to a self-transition.

Regular PDDL goals are used to express goal-type performance objectives. A goal statement `(:goal  $\phi$ )` for a probabilistic planning problem encodes the objective that the probability of achieving  $\phi$  should be maximized, unless an explicit optimization metric is specified for the planning problem.

For planning problems instantiated from a domain declaring the `:rewards` requirement, the default plan objective is to maximize the expected reward. A goal statement in the specification of a reward oriented planning problem identifies a set of absorbing states. In addition to transition rewards specified in action effects, it is possible to associate a one-time reward for entering a goal state. This is done using the `(:goal-reward  $f$ )` construct, where  $f$  is a numeric expression.

In general, a statement `(:metric maximize  $f$ )` in a problem definition means that the expected value of  $f$  should be maximized. PPDDL defines `goal-probability` as a special optimization metric that can be used to explicitly specify that the plan objective is to maximize (or minimize) the probability of goal achievement.

## Formal Semantics

We present a formal semantics for PPDDL planning problems in terms of a mapping to a probabilistic transition system with rewards. A planning problem defines a set of state variables  $V$ , possibly containing both Boolean and numeric state variables. An assignment of values to state variables defines a state, and the state space  $S$  of the planning problem is the set of states representing all possible assignments of values to variables. In addition to  $V$ , a planning prob-

```

(define (domain coffee-delivery)
  (:requirements :negative-preconditions :disjunctive-preconditions
    :conditional-effects :mdp)
  (:predicates (in-office) (raining) (has-umbrella) (is-wet)
    (has-coffee) (user-has-coffee))
  (:action buy-coffee
    :effect (and (when (not (in-office)) (probabilistic 0.8 (has-coffee)))
      (when (user-has-coffee) (increase (reward) 0.8))
      (when (not (is-wet)) (increase (reward) 0.2))))
  ...))

```

Figure 2: Part of PPDDL encoding of “Coffee Delivery” domain.

lem defines an initial-state distribution  $p_0 : S \rightarrow [0, 1]$  with  $\sum_{s \in S} p_0(s) = 1$  (i.e.  $p_0$  is a probability distribution over states), a formula  $\phi$  over  $V$  characterizing a set of goal states  $G = \{s \mid s \models \phi\}$ , a one-time reward  $r_G$  associated with entering a goal state, and a set of actions  $A$  instantiated from PPDDL action schemata. For goal-directed planning problems, without explicit rewards, we use  $r_G = 1$ .

An action  $a \in A$  consists of a precondition  $\phi_a$  and an effect  $e_a$ . Action  $a$  is applicable in a state  $s$  if and only if  $s \models \phi_a$ . It is an error to apply  $a$  to a state such that  $s \not\models \phi_a$ . This is consistent with the semantics of PDDL2.1 (Fox & Long 2003) and permits the modeling of forced chains of actions. Effects are recursively defined as follows (cf. Rintanen 2003):

1.  $\top$  is the null-effect, represented in PPDDL by  $(\text{and})$ .
2.  $b$  and  $\neg b$  are effects if  $b \in V$  is a Boolean state variable.
3.  $x \leftarrow f$  is an effect if  $x \in V$  is a numeric state variable and  $f$  is a real-valued function on numeric state variables.
4.  $r \uparrow f$  is an effect if  $f$  is a real-valued function on numeric state variables.
5.  $e_1 \wedge \dots \wedge e_n$  is an effect if  $e_1, \dots, e_n$  are effects.
6.  $c \triangleright e$  is an effect if  $c$  is a formula over  $V$  and  $e$  is an effect.
7.  $p_1 e_1 \mid \dots \mid p_n e_n$  is an effect if  $e_1, \dots, e_n$  are effects,  $p_i \geq 0$  for all  $i \in \{1, \dots, n\}$ , and  $\sum_{i=1}^n p_i = 1$ .

Items 2 through 4 are referred to as *simple effect*. The effect  $b$  sets the Boolean state variable  $b$  to true in the next state, while  $\neg b$  sets  $b$  to false in the next state. For  $x \leftarrow f$ , the value of  $f$  in the current state becomes the value of the numeric state variable  $x$  in the next state. Effects of the form  $r \uparrow f$  are used to associate rewards with transitions as described below.

An action  $a = \langle \phi_a, e_a \rangle$  defines a transition probability matrix  $P_a$  and a transition reward matrix  $R_a$ , with  $p_{ij}^a$  being the probability of transitioning to state  $j$  when applying  $a$  in state  $i$ , and  $r_{ij}^a$  being the reward associated with the state transition from  $i$  to  $j$  when caused by  $a$ . We can compute  $P_a$  and  $R_a$  by first translating  $e_a$  into an effect of the form  $p_1 e_1 \mid \dots \mid p_n e_n$ , where each  $e_i$  is a deterministic effect. Rintanen (2003) calls this form Unary Nondeterminism Normal Form. Any effect  $e$  can be translated into this form by using the top four equivalences in Figure 3.

We further rewrite the effect of an action by translating each  $e_i$  into an effect of the form  $(c_{i1} \triangleright e_{i1}) \wedge \dots \wedge (c_{in_i} \triangleright$

$e_{in_i})$ , where each  $e_{ij}$  is a conjunction of simple effects and the conditions are mutually exclusive and exhaustive (i.e.  $c_{ij} \wedge c_{ik} \equiv \perp$  for all  $j \neq k$  and  $\bigvee_{j=1}^{n_i} c_{ij} \equiv \top$ ). The bottom four equivalences in Figure 3 allow us to perform the desired translation.

An effect of the form  $c \triangleright e$ , where  $e$  is a conjunction of simple effects, defines a set of state transitions. We assume that  $e$  is consistent. Actions with inconsistent effects are not valid PPDDL actions, and care should be taken when designing a PPDDL domain to ensure that no instantiations of action schemata can have inconsistent effects. A conjunction of simple effects is inconsistent if it contains both  $b$  and  $\neg b$ , or multiple *non-commutative* updates of a single numeric state variable. Two effects  $x \leftarrow f$  and  $x \leftarrow f'$  are commutative if  $f(s[x = f'(s)]) = f'(s[x = f(s)])$ , where  $f(s)$  is the value of  $f$  evaluated in state  $s$  and  $s[x = y]$  denotes a state with all state variables having the same value as in state  $s$ , except for  $x$  which has value  $y$ , i.e. numeric effects are commutative if they are insensitive to ordering. Under these assumptions, the following function can be defined:

$$\begin{aligned}
\tau(s, s', \top) &\doteq s' \\
\tau(s, s', b) &\doteq s'[b = \top] \\
\tau(s, s', \neg b) &\doteq s'[b = \perp] \\
\tau(s, s', x \leftarrow f) &\doteq s'[x = f(s)] \\
\tau(s, s', r \uparrow f) &\doteq s' \\
\tau(s, s', e_1 \wedge e_2) &\doteq \tau(s, \tau(s, s', e_1), e_2)
\end{aligned}$$

We can use  $\tau$  to describe the set of state transitions defined by the effect  $c \triangleright e$ :

$$T(c \triangleright e) = \{\langle s, s' \rangle \mid s \models c \text{ and } s' = \tau(s, s, e)\}.$$

Given this definition of  $T(c \triangleright e)$ , we can compute a transition matrix  $T_{ij}$  for each  $c_{ij} \triangleright e_{ij}$ . The element at row  $s$  and column  $s'$  of  $T_{ij}$  is 1 if  $\langle s, s' \rangle \in T(c_{ij} \triangleright e_{ij})$ , and 0 otherwise. Since we have ensured that the conditions  $c_{ij}$  are mutually exclusive, we get  $P_a = \sum_{i=1}^n p_i T_i$  as the transition probability matrix for action  $a$ , where  $T_i = \sum_{j=1}^{n_i} T_{ij}$ . Finally, we need to make all states that satisfy the goal condition  $\phi$  of the problem absorbing. This is accomplished by modifying  $P_a$ : for each  $s$  such that  $s \models \phi$ , we set the entry at row  $s$  and column  $s$  to 1 and the remaining entries on the same row to 0.

$$\begin{aligned}
e &\equiv 1e \\
e \wedge (p_1 e_1 | \dots | p_k e_n) &\equiv p_1(e \wedge e_1) | \dots | p_n(e \wedge e_n) \\
c \triangleright (p_1 e_1 | \dots | p_n e_n) &\equiv p_1(c \triangleright e_1) | \dots | p_n(c \triangleright e_n) \\
p_1(p'_1 e'_1 | \dots | p'_k e'_k) | p_2 e_2 | \dots | p_n e_n &\equiv (p_1 p'_1) e_1 | \dots | (p_1 p'_k) e'_k | p_2 e_2 | \dots | p_n e_n \\
e &\equiv \top \triangleright e \\
c \triangleright e &\equiv (c \triangleright e) \wedge (\neg c \triangleright \top) \\
c \triangleright (c' \triangleright e) &\equiv (c \wedge c') \triangleright e \\
(c_1 \triangleright e_1) \wedge (c_2 \triangleright e_2) &\equiv ((c_1 \wedge c_2) \triangleright (e_1 \wedge e_2)) \wedge ((c_1 \wedge \neg c_2) \triangleright e_1) \\
&\quad \wedge ((\neg c_1 \wedge c_2) \triangleright e_2) \wedge ((\neg c_1 \wedge \neg c_2) \triangleright \top)
\end{aligned}$$

Figure 3: Effect equivalences.

The reward associated with a conjunction of simple effects can be defined as follows:

$$\begin{aligned}
r(s, \top) &\doteq 0 \\
r(s, b) &\doteq 0 \\
r(s, \neg b) &\doteq 0 \\
r(s, x \leftarrow f) &\doteq 0 \\
r(s, r \uparrow f) &\doteq f(s) \\
r(s, e_1 \wedge e_2) &\doteq r(s, e_1) + e(s, e_2)
\end{aligned}$$

The effect  $c_{ij} \triangleright e_{ij}$  associates reward  $r(s, e_{ij})$  with each transition  $\langle s, s' \rangle \in T(c_{ij} \triangleright e_{ij})$ . We define a transition reward matrix  $R_{ij}$  for  $c_{ij} \triangleright e_{ij}$ . The element at row  $s$  and column  $s'$  of  $R_{ij}$  is  $r(s, e_{ij})$  for  $s' = \tau(s, s, e_{ij})$  and 0 if  $\langle s, s' \rangle \notin T_{ij}$ . We then sum over all  $c_{ij} \triangleright e_{ij}$  to get a transition reward matrix for  $e_i$ :  $R_i = \sum_{j=1}^{n_i} R_{ij}$ .

The same transition may occur in multiple outcomes of the effect  $p_1 e_1 | \dots | p_n e_n$ , and we require the reward for a specific transition to be consistent across outcomes. Let  $\bullet$  represent the fact that the reward is undefined for a transition. We define  $\tilde{R}_i$  to be  $R_i$  with an element at row  $s$  and column  $s'$  set to  $\bullet$  if the element at row  $s$  and column  $s'$  of  $T_i$  is zero (i.e.  $e_i$  does not define a transition from  $s$  to  $s'$ ). We define an element-wise matrix operator  $\odot$  as follows:

$$\begin{aligned}
\bullet \odot x &\doteq x \\
x \odot \bullet &\doteq x \\
x \odot x &\doteq x \\
x \odot y &\doteq \text{error if } x \neq y
\end{aligned}$$

We can now define the transition reward matrix for action  $a$ :  $R_a = R_G + \odot_{i=1}^n \tilde{R}_i$ .  $R_G$  represents the one-time reward associated with goal states. The entry at row  $s$  and column  $s'$  of  $R_G$  is set to  $r_G$  if  $s \not\models \phi$  and  $s' \models \phi$ , and 0 otherwise. The transition reward matrix is well-defined if and only if the transition rewards are consistent across all outcomes of an action.

## References

Dean, T., and Kanazawa, K. 1989. A model for reasoning about persistence and causation. *Computational Intel-*

*ligence* 5(3):142–150.

Dearden, R., and Boutilier, C. 1997. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence* 89(1–2):219–283.

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.

Ghallab, M.; Howe, A. E.; Knoblock, C. A.; McDermott, D.; Ram, A.; Veloso, M. M.; Weld, D. S.; and Wilkins, D. 1998. PDDL—the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, CT.

Howard, R. A. 1960. *Dynamic Programming and Markov Processes*. New York, NY: John Wiley & Sons.

Kushmerick, N.; Hanks, S.; and Weld, D. S. 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76(1–2):239–286.

Littman, M. L. 1997. Probabilistic propositional planning: Representations and complexity. In *Proc. Fourteenth National Conference on Artificial Intelligence*, 748–754. Providence, RI: American Association for Artificial Intelligence.

McDermott, D. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–55.

Rintanen, J. 2003. Expressive equivalence of formalism for planning with sensing. In Giunchiglia, E.; Muscettola, N.; and Nau, D. S., eds., *Proc. Thirteenth International Conference on Automated Planning and Scheduling*, 185–194. Trento, Italy: AAAI Press.

# mGPT: A Probabilistic Planner based on Heuristic Search

**Blai Bonet**

Departamento de Computación  
Universidad Simón Bolívar  
Caracas, Venezuela  
bonet@ldc.usb.ve

**Héctor Geffner**

Departament de Tecnologia  
Universitat Pompeu Fabra  
Barcelona 08003, España  
hector.geffner@tecn.upf.es

## Abstract

We describe the version of the GPT planner to be used in the planning competition. This version, called mGPT, solves MDPs specified in the PPDDL language by extracting and using different classes of lower bounds, along with various heuristic-search algorithms. The lower bounds are extracted from deterministic relaxations of the MDP where alternative probabilistic effects of an action are mapped into different, independent, deterministic actions. The heuristic-search algorithms, on the other hand, use these lower bounds for focusing the updates and delivering a consistent value function over all states reachable from the initial state with the greedy policy.

## Introduction

mGPT is a planner based on heuristic search for solving MDP models specified in the high-level planning language PPDDL. mGPT captures a fragment of the functionality of the GPT system that features non-determinism and incomplete information, in both qualitative and probabilistic forms, like POMDPs and Conformant planning (Bonet & Geffner 2001a; Bonet & Thiébaux 2003).

mGPT supports several algorithms and heuristic functions (lower bounds) that when combined generate a wide range of different solvers. The two main algorithms are `lrtdp` and `hdp`. Both are heuristic-search algorithms that make use of a given initial state  $s_0$  and lower bound information. More precisely, they compute a value function  $V$  with a residual bounded by a user-provided threshold over all states reachable from  $s_0$  when using the greedy policy  $\pi_V$  (Bonet & Geffner 2003b; 2003a).

The lower bounds are derived by solving relaxations of the input problem with an algorithms provided by mGPT. Since these algorithms are also based on heuristic search, we have implemented “stackable” components that are created in sequence for computing complex heuristic functions from simpler ones.

In this short document, we describe the features of the mGPT planner. This document is organized as follows. In the following two sections, we give a

brief description of the most important algorithms and heuristics functions implemented in mGPT. Then, we describe how these algorithm and heuristics can combined in order to generate a wide range of different solvers. We conclude with a short discussion.

## Algorithms

We divide the algorithms in two groups of optimal and suboptimal algorithms.

An optimal algorithm is one that computes an  $\epsilon$ -consistent value function  $V$  over all states reachable from the initial state  $s_0$  with the greedy policy with respect to  $V$ , denoted as  $\pi_V$ . A value function  $V$  is  $\epsilon$ -consistent at state  $s$  if its residual at  $s$  is less than or equal to  $\epsilon$ . It is known that if  $V$  is 0-consistent over all states reachable from  $s_0$  with  $\pi_V$ , then  $\pi_V$  is optimal, as well as if  $V$  is  $\epsilon$ -consistent for a sufficiently small  $\epsilon$ . Here  $\epsilon$  is a user-provided parameter.

The suboptimal algorithms, on the other hand, are provided in order to interleave planning and execution. In this group, we include algorithms that start selecting actions with respect to an initial lower bound (heuristic) that is improved over time.

(Although our main interest is towards optimal algorithms, we have included the suboptimal ones in order to cope with the format of the competition.)

The main optimal algorithms are `vi`, `lrtdp` and `hdp`, whilst the suboptimal ones are `asp` and `hdp-i`. In the following, we give a brief description and references for these algorithms.

The Value Iteration algorithm (`vi`) solves the problem in two steps. First, it generates the reachable state space from the initial state and the applicable operators, and second, uses the Value Iteration algorithm to obtain an optimal solution for the problem. `vi` is included in mGPT as a bottom-line reference.

Labeled Real-Time Dynamic Programming (`lrtdp`) is a heuristic-search algorithm that implements a labeling scheme on top of the `rtdp` algorithm (Barto, Bradtke, & Singh 1995). `lrtdp` works by performing simulated trials that start at the initial state and end at “solved” states by selecting actions with respect to  $\pi_V$ . Initially,  $V$  is the input heuristic function, and

the only solved states are the goal states. Then, each time an action is picked at state  $s$ , the value of  $s$  is updated by making its value consistent with the value of its successor states. At the end of each trial, a labeling procedure is called that checks whether new states can be labeled as solved: a state is solved if its value and the value of all its descendents are  $\epsilon$ -consistent. The algorithm ends when the initial state is labeled solved since, at that time, all states reachable from  $s_0$  with  $\pi_V$  are consistent. As shown in (Bonet & Geffner 2003b), this labeling mechanism adds a crisp termination condition to **rtdp** that features faster convergence time while retaining its good anytime behavior.

Since  $\pi_V$ , the policy returned by **lrrtdp**, is only guaranteed to be optimal over a subset of states, i.e.  $s_0$  and those reachable from it, then  $\pi_V$  is said to be a partial optimal policy *closed with respect to*  $s_0$ .

Heuristic Dynamic Programming (**hdp**) is also a heuristic-search algorithm that computes a partial optimal policy closed with respect to  $s_0$ . The **hdp** algorithm works by performing depth-first searches in state space looking for  $\epsilon$ -inconsistent states, and then updating their values to make them consistent. The searches are stopped when no inconsistent states are found (Bonet & Geffner 2003a).

Action Selection for Planning (**asp**) is a reactive algorithm that starts by selecting actions with respect to the input heuristic function. Each time an action is needed for state  $s$ , **asp** performs multiple depth-bounded **rtdp**-like trials starting at  $s$  before returning an action for  $s$ . These simulations implement a bounded-lookahead mechanism that improve the action selection task. This **asp** algorithm is a generalization of (Bonet, Loerincs, & Geffner 1997) for probabilistic planning.

Approximated Heuristic DP (**hdp-i**) is a heuristic-search algorithm that like **hdp** performs searches and updates. Unlike **hdp**, the **hdp-i** algorithm only enforces consistency over all states reachable from  $s_0$  with plausibility no smaller than  $i$ . These plausibility levels form a *qualitative* scale based on *kappa* rankings (Spohn 1988; Pearl 1993) that quantify how improbable is to make a transition from the initial state to the given state. The **hdp-i** algorithm and some of its properties are described in (Bonet & Geffner 2003a).

## Heuristics

The heuristics functions are also divided in two groups of admissible and non-admissible heuristics. An admissible heuristic is one that never overestimates the optimal cost, i.e. a lower bound. The main admissible heuristics are **zero**, **min-min**, **atom-min-forward** and **atom-min-backward**, whilst the main non-admissible heuristic is **ff**. All these heuristic are computed by solving deterministic relaxations of the input problem. In the case of admissible heuristics, these relaxations must be solved optimally (Pearl 1983).

The most important relaxations are the weak and

strong relaxations. The weak relaxation is computed by transforming the input problem into a deterministic problem in which every operator of the form

$$\langle \text{prec}, [p_1 : \alpha_1, \dots, p_n : \alpha_n] \rangle, \quad (1)$$

where **prec** is the precondition and  $\alpha_i$  is the  $i$ -th effect with probability  $p_i$ , is translated into the  $n$  deterministic and independent operators  $\langle \text{prec}, \alpha_i \rangle$ .

It is not hard to show that the optimal solution for the weak relaxation is a lower bound one the optimal solution for the original problem.

The strong relaxation is a STRIPS problem computed by firstly transforming the input into a problem in which every operator is of the form

$$\langle \text{prec}, [p_1 : (\text{add}_1, \text{del}_1), \dots, p_n : (\text{add}_n, \text{del}_n)] \rangle \quad (2)$$

where **prec**, **add**<sub>1</sub>, ..., **del** <sub>$n$</sub>  are all *conjunctions* of literals and  $\sum_i p_i = 1$ . Observe that in order to take the input problem into the form given by (2), we must remove disjunctive preconditions, conditional effects, quantifier symbols, etc. The strong relaxation is then generated by translating each operator (2) into the  $n$  deterministic and independent STRIPS operators

$$\langle \text{prec}, \text{add}_i, \text{del}_i \rangle. \quad (3)$$

As before, it is not hard to show that the optimal solution for the strong relaxation is a lower bound on the optimal solution for the original problem.

In the following, we give a brief description of the different heuristic and their relation to the relaxations.

The Min-Min (**min-min**) heuristic is the optimal solution to the deterministic problem given by the weak relaxation. Two flavors are provided: **min-min-lrrtdp** that solves the relaxation with a deterministic version of **lrrtdp** (a.k.a. **lrta** (Korf 1990)), and **min-min-ida\*** that solves the relaxation with IDA\*. Both versions are *lazy* in the sense that the values are computed on a need basis as the planner requires them. See (Bonet & Geffner 2003b; 2003a) for references. (Since the **min-min** heuristic is computed with a heuristic-search algorithm, another heuristic function is required for its computation. Below, we describe how to specify these multiple heuristics.)

Atom-Min Forward (**atom-min-forward**) is a heuristic function computed in *atom space* from the strong relaxation. **atom-min-forward** computes “costs” of reaching set of atoms of fixed cardinality from a given state. The name forward comes from the fact that the costs are computed by a forward-chaining procedure that begins with the given state and ends when the goal is generated. This heuristic is a generalization of the  $h_{\min}$  heuristic in HSP (Bonet & Geffner 2001b). As in **min-min**, the heuristic values are computed on demand. **atom-min-k-forward** refers to the **atom-min-forward** heuristic for sets of cardinality  $k$ . The **atom-min-forward** heuristic is from (Haslum & Geffner 2000).

Atom-Min Backward (**atom-min-backward**) is a heuristic similar to **atom-min-forward** except that it

computes costs of reaching sets of atoms from the *goal* state in an inverted version of the strong relaxation. Thus, before the search starts, all costs for all sets of atoms of fixed cardinality are computed and stored in a table that are later used to compute the heuristic function. The inverted relaxation is described in (Bonet & Geffner 2001b).

The FF (**ff**) heuristic implements the heuristic function used in the FF planner with respect to the strong relaxation (Hoffmann & Nebel 2001). This heuristic is informative but non-admissible and can only be used for non-optimal planning.

## Combining Algorithms and Heuristics

The main parameters for mGPT are “-p <planner>” that specify the algorithm to use for the planner, “-h <heuristics>” that specify the heuristic function, and “-e <epsilon>” that specify the threshold  $\epsilon$  for the consistency check.

One typical call looks like

```
mGPT -p lrtdp \
      -h "atom-min-1-forward" \
      -e .001 <rest>
```

which instructs mGPT to use the `lrtdp` algorithm with the `atom-min-1-forward` heuristic and  $\epsilon = 0.001$ . Since the algorithm is optimal and the heuristic is admissible, this call produces an optimal policy. The `atom-min-1-forward` heuristic is admissible but very weak. The following example shows how to use the `min-min-lrtdp` heuristic using `atom-min-1-forward` as the base heuristic:

```
mGPT -p lrtdp \
      -h "atom-min-1-forward|min-min-lrtdp" \
      -e .001 <rest>
```

Note how the pipe symbol is used to stack the components of the heuristic function.

Another possibility is to use mGPT as a reactive planner in which decisions are taken on-line with respect to a heuristic function that is improved over time. For example,

```
mGPT -p asp -h "ff" <rest>
```

uses the `asp` algorithm with the `ff` heuristic, while

```
mGPT -p asp -h "zero|min-min-ida*" \
      -e .001 <rest>
```

uses the `asp` algorithm with the `min-min-ida*` heuristic computed from the constant-zero heuristic. In the first case, the heuristic being used is non-admissible, so the planner will deliver a suboptimal policy. In the latter case, the `asp` algorithm is seeded with an admissible heuristic so it is guaranteed to converge to a partial optimal policy as the number of trials increase.

Other combinations of algorithms and heuristics are possible. mGPT also implements other heuristic functions and parameters to control number of simulation trials and cutoff length for `asp`, initial hash size, heuristic weight, dead-end value, verbosity level, etc.

## Discussion

At the moment of writing these pages, it is not clear for us which combination of algorithm and heuristic is going to be used during the competition. Moreover, we could enter the competition either with a fixed choice, or with a more complex planner that picks a choice upon an analysis of the input problem. In any case, we plan to evaluate (after the competition) the different choices separately in order to obtain meaningful data for future research.

The mGPT planner will be publicly available after the competition with the default settings corresponding to those actually used.

**Acknowledgements** We thank the chairs of IPC-4 for making this competition possible. mGPT was built upon a source code developed by John Asmuth from CMU and distributed by the organizers.

## References

- Barto, A.; Bradtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72:81–138.
- Bonet, B., and Geffner, H. 2001a. GPT: a tool for planning with uncertainty and partial information. In *Proc. IJCAI/Workshop on Planning with Uncertainty and Partial Information*, 82–87.
- Bonet, B., and Geffner, H. 2001b. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.
- Bonet, B., and Geffner, H. 2003a. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *Proc. IJCAI-03*, 1233–1238.
- Bonet, B., and Geffner, H. 2003b. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. ICAPS-03*, 12–21.
- Bonet, B., and Thiébaux, S. 2003. GPT meets PSR. In *Proc. ICAPS-03*, 102–111.
- Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *Proc. AAAI-97*, 714–719.
- Haslum, P., and Geffner, H. 2000. Admissible heuristic for optimal planning. In *Proc. AIPS-2000*, 140–149.
- Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.
- Korf, R. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2–3):189–211.
- Pearl, J. 1983. *Heuristics*. Morgan Kaufmann.
- Pearl, J. 1993. From conditional oughts to qualitative decision theory. In *Proc. UAI-93*, 12–22.
- Spohn, W. 1988. A general non-probabilistic theory of inductive reasoning. In *Proc. UAI-88*, 149–158.

# Symbolic Heuristic Search for Probabilistic Planning

**Zhengzhu Feng**

Department of Computer Science  
University of Massachusetts  
Amherst, MA 01003  
fengzz@cs.umass.edu

**Eric A. Hansen**

Department of Computer Science and Engineering  
Mississippi State University  
Mississippi State, MS 39762  
hansen@cse.msstate.edu

## Abstract

We describe a planner that participates in the Probabilistic Planning Track of the 2004 International Planning Competition. Our planner integrates two approaches to solving Markov decision processes with large state spaces. State abstraction is used to avoid evaluating states individually. Forward search from a start state, guided by an admissible heuristic, is used to avoid evaluating all states.

## Introduction

The 2004 International Planning Competition introduces, for the first time, a probabilistic planning track. The underlying model of the planning problem is essentially a Markov decision process (MDP), and is encoded using an extension of the PDDL language, called the Probabilistic PDDL. Classic dynamic programming algorithms solve MDPs in time polynomial in the size of the state space. However, the size of the state space grows exponentially with the number of features describing the problem. This “state explosion” problem limits use of the MDP framework, and overcoming it has become an important topic of research.

Over the past several years, approaches to solving MDPs that do not rely on complete state enumeration have been developed. One approach exploits a feature-based (or factored) representation of an MDP to create state abstractions that allow the problem to be represented and solved more efficiently (Dearden & Boutilier 1997; Hoey et al. 1999; and many others). Another approach limits computation to states that are reachable from the starting state(s) of the MDP (Barto, Bradtke, & Singh 1995; Dean *et al.* 1995; Hansen & Zilberstein 2001). Our planner integrates these approaches in a unifying framework using symbolic model-checking techniques, based on the symbolic LAO\* and symbolic RTDP algorithms we previously developed (Feng & Hansen 2002; Feng, Hansen, & Zilberstein 2003). In this paper we present a brief summary of these algorithms.

## Factored MDPs and decision diagrams

A Markov decision process (MDP) is defined as a tuple  $(S, A, P, R)$  where:  $S$  is a set of states;  $A$  is a set of actions;  $P$  is a set of transition models  $P^a : S \times S \rightarrow [0, 1]$ , one for each action, specifying the transition probabilities of the process; and  $R$  is a set of reward models  $R^a : S \rightarrow \mathbb{R}$ ,

one for each action, specifying the expected reward for taking action  $a$  in each state. We consider MDPs for which the objective is to find a policy  $\pi : S \rightarrow A$  that maximizes total discounted reward over an infinite (or indefinite) horizon, where  $\gamma \in [0, 1]$  is the discount factor. (We allow a discount factor of 1 for indefinite-horizon problems only, that is, for MDPs that terminate after a goal state is reached.)

In a factored MDP, the set of states is described by a set of random variables  $\mathbf{X} = \{X_1, \dots, X_n\}$ . Without loss of generality, we assume these are Boolean variables. A particular instantiation of the variables corresponds to a unique state. Because the set of states  $S = 2^{\mathbf{X}}$  grows exponentially with the number of variables, it is impractical to represent the transition and reward models explicitly as matrices when the number of state variables is large. Instead we follow Hoey *et al.* (1999) in using algebraic decision diagrams to achieve a more compact representation.

Algebraic decision diagrams (ADDs) are a generalization of binary decision diagrams (BDDs), a compact data structure for Boolean functions used in symbolic model checking. A decision diagram is a data structure (corresponding to an acyclic directed graph) that compactly represents a mapping from a set of Boolean state variables to a set of values. A BDD represents a mapping to the values 0 or 1. An ADD represents a mapping to any finite set of values. To represent these mappings compactly, decision diagrams exploit the fact that many instantiations of the state variables map to the same value. In other words, decision diagrams exploit state abstraction. BDDs are typically used to represent the characteristic functions of sets of states and the transition functions of finite-state automata. ADDs can represent weighted finite-state automata, where the weights correspond to transition probabilities or rewards, and thus are an ideal representation for MDPs.

Hoey *et al.* (1999) describe how to represent the transition and reward models of a factored MDP compactly using ADDs. We adopt their notation and refer to their paper for details of this representation. Let  $\mathbf{X} = \{X_1, \dots, X_n\}$  represent the state variables at the current time and let  $\mathbf{X}' = \{X'_1, \dots, X'_n\}$  represent the state variables at the next step. For each action, an ADD  $P^a(\mathbf{X}, \mathbf{X}')$  represents the transition probabilities for the action. Similarly, the reward model  $R^a(\mathbf{X})$  for each action  $a$  is represented by an ADD. The advantage of using ADDs to represent mappings from states

(and state transitions) to values is that the complexity of operators on ADDs depends on the number of nodes in the diagrams, not the size of the state space. If there is sufficient regularity in the model, ADDs can be very compact, allowing problems with large state spaces to be represented and solved efficiently.

### Symbolic LAO\* algorithm

LAO\* (Hansen & Zilberstein 2001) is an extension of the classic search algorithm AO\* that can find solutions with loops. This makes it possible for LAO\* to solve MDPs, since a policy for an infinite-horizon MDP allows both conditional and cyclic behavior. Like AO\*, LAO\* has two alternating phases. First, it expands the best partial solution (or policy) and evaluates the states on its fringe using an admissible heuristic function. Then it performs dynamic programming on the states visited by the best partial solution, to update their values and possibly revise the currently best partial solution. The two phases alternate until a complete solution is found, which is guaranteed to be optimal.

AO\* and LAO\* differ in the algorithms they use in the dynamic programming step. Because AO\* assumes an acyclic solution, it can perform dynamic programming in a single backward pass from the states on the fringe of the solution to the start state. Because LAO\* allows solutions with cycles, it relies on an iterative dynamic programming algorithm (such as value iteration or policy iteration). In organization, the LAO\* algorithm is similar to the “envelope” dynamic programming approach to solving MDPs (Dean *et al.* 1995). It is also closely related to RTDP (Barto, Bradtke, & Singh 1995), which is an on-line (or “real time”) search algorithm for MDPs, in contrast to LAO\*, which is an off-line search algorithm.

We call our generalization of LAO\* a symbolic search algorithm because it manipulates sets of states, instead of individual states. In keeping with the symbolic model-checking approach, we represent a set of states  $S$  by its characteristic function  $\chi_S$ , so that  $s \in S \iff \chi_S(s) = 1$ . We represent the characteristic function of a set of states by an ADD. (Because its values are 0 or 1, we can also represent a characteristic function by a BDD.) From now on, whenever we refer to a set of states,  $S$ , we implicitly refer to its characteristic function, as represented by a decision diagram.

In addition to representing sets of states as ADDs, we represent every element manipulated by the LAO\* algorithm as an ADD, including: the transition and reward models; the policy  $\pi : S \rightarrow A$ ; the state evaluation function  $V : S \rightarrow \mathbb{R}$  that is computed in the course of finding a policy; and an admissible heuristic evaluation function  $h : S \rightarrow \mathbb{R}$  that guides the search for the best policy. Even the discount factor  $\gamma$  is represented by a simple ADD that maps every input to a constant value. This allows us to perform all computations of the LAO\* algorithm using ADDs.

Besides exploiting state abstraction, we want to limit computation to the set of states that are reachable from the start state by following the best policy. Although an ADD effectively assigns a value to every state, these values are only relevant for the set of reachable states. To focus computation on the relevant states, we introduce the notion of

*masking* an ADD. Given an ADD  $D$  and a set of relevant states  $U$ , masking is performed by multiplying  $D$  by  $\chi_U$ . This has the effect of mapping all irrelevant states to the value zero. We let  $D_U$  denote the resulting *masked ADD*. (Note that we need to have  $U$  in order to correctly interpret  $D_U$ .) Mapping all irrelevant states to zero can simplify the ADD considerably. If the set of reachable states is small, the masked ADD often has dramatically fewer nodes. This in turn can dramatically improve the efficiency of computation using ADDs.

Symbolic LAO\* does not maintain an explicit search graph. It is sufficient to keep track of the set of states that have been “expanded” so far, denoted  $G$ , the *partial value function*, denoted  $V_G$ , and a *partial policy*, denoted  $\pi_G$ . For any state in  $G$ , we can “query” the policy to determine its associated action, and compute its successor states. Thus, the graph structure is implicit in this representation. Note that throughout the whole LAO\* algorithm, we only maintain one value function  $V$  and one policy  $\pi$ .  $V_G$  and  $\pi_G$  are implicitly defined by  $G$  and the masking operation.

### Symbolic RTDP

Recall that RTDP performs a DP update while interacting with the environment. At each time step  $t$ , the agent observes the current state  $s_t$  and performs a DP backup to update its value, as follows:

$$V^{t+1}(s_t) \leftarrow \max_{a \in A} \left\{ R^a(s_t) + \gamma \sum_{s' \in S} P^a(s_t, s') V^t(s') \right\}. \quad (1)$$

The values of all other states are kept unchanged, that is, for all  $s \neq s_t$ :

$$V^{t+1}(s) = V^t(s).$$

If the initial value function is an admissible estimate of the optimal value function, then an agent can always take the action that maximizes Equation (1). Otherwise some exploration scheme must be used in choosing actions, in order to ensure convergence. After an action is taken, the agent observes the resulting state and the cycle repeats.

The advantage of RTDP over standard DP is that it uses an on-line trajectory of states, beginning from the start state, to determine what states to update and to avoid computations on unlikely states. However, the enumerative nature of the trajectory sampling is a bottleneck for further performance improvement. When the state space is large enough, a state by state update becomes hopelessly inefficient, especially if the sampling involves carrying out physical actions. sRTDP helps overcome this inefficiency by generalizing the update from a single state to an abstract state, using symbolic model checking techniques.

We extend the idea of masking in symbolic LAO\* to sRTDP by performing DP on the abstract state  $E$  that the current state  $s$  belongs to. Symbolic model-checking provides us with convenient and efficient techniques to group states as abstract states and to manipulate these abstract states. There are many ways to group states into abstract states. We present two heuristic approaches that are motivated by the idea of generalization by structural similarity. A



*value-based* abstract state consists of states whose value estimates are close to that of the current state. A *reachability-based* abstract state consists of states that share with the current state a similar set of successor states. Unlike SPUDD, we *explicitly* construct this abstract state at each time step of sRTDP, using standard ADD model-checking operators.

**Generalization by Value** With a value-based abstract state, the experience is generalized to states that have similar value estimates as the current state. The intuition is that states with similar *optimal* values may also be similarly desirable. Generalizing updates to states with similar *estimated* values helps the agent in two ways. First, if some of these states indeed have similar optimal value as the current state, the update strengthens this similarity and the agent is better informed in the future when these states are visited again. Second, if some of the states have very different optimal value than the current state, the generalization helps to distinguish them and avoid computations on them in the future when the same state as the current state is visited again.

**Generalization by Reachability** With a reachability-based abstract state, the experience is generalized to states that are similar to the current state in terms of the set of one-step reachable states. The intuition here is that if the agent is going to visit some states, say  $C$ , from the current state  $s$ , then any information about  $C$  is useful not only to  $s$  but also to other states that can reach  $C$ . By generalizing the update to these other states the agent is better informed in the future whether to aim at  $C$  or to avoid it.

To compute the abstract state based on reachability, we introduce two operators from the model-checking literature. The  $Img(C)$  operator computes the set of one-step reachable states from states in  $C$ , and the  $PreImg(C)$  operator computes the set of states that can reach some state in  $C$  in one step. The reachability-based abstract state  $E$  can then be computed as:

$$E = PreImg(Img(\{s\})) - PreImg(S - Img(\{s\})).$$

Once the set  $E$  is computed, it is used to mask the current value function before performing the DP update. After the update, an action is chosen that maximizes the DP update at state  $s$ . The agent then carries out the action, and the process repeats.

Although both symbolic LAO\* and sRTDP use a “masked” DP update, the masks they use are different and serve different purposes. The mask in symbolic LAO\* contains all states visited so far by the forward search step. The purpose of masking is to restrict computation to relevant states. The mask in sRTDP contains states that share structural similarity. The purpose of masking is to generalize update on a single state to an abstract state. This generalization has two consequences. It introduces some overhead in the DP step, including identifying the abstract state, and performing masked DP instead of single-state DP. On the other hand, it updates the value of a group of states in a single step, at a cost that can be significantly less than updating the states separately. For problems that are large enough yet have special

## Admissible heuristics

Both LAO\* and (model-based) RTDP use an admissible heuristic to guide the search. From the initial release of the sample test problems from the planning competition, it is possible to design domain specific heuristic functions. On the other hand, if such a heuristic is not available, we can always revert to a simple heuristic using approximate dynamic programming. Given an error bound on the approximation, the value function can be converted to an admissible heuristic. (Another way to ensure admissibility is to perform value iteration on an initial value function that is admissible, since each step of value iteration preserves admissibility.) Symbolic dynamic programming can be used to compute an approximate value function efficiently. St. Aubin et al. (2000) describe an approximate dynamic programming algorithm for factored MDPs, called APRICODD, that is based on SPUDD. It simplifies the value function ADD by aggregating states with similar values. Another approach to approximate dynamic programming for factored MDPs described by Dearden and Boutilier (1997) can also be used to compute admissible heuristics.

## References

- Barto, A.; Bradtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72:81–138.
- Dean, T.; Kaelbling, L.; Kirman, J.; and Nicholson, A. 1995. Planning under time constraints in stochastic domains. *Artificial Intelligence* 76:35–74.
- Dearden, R., and Boutilier, C. 1997. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence* 89:219–283.
- Feng, Z., and Hansen, E. A. 2002. Symbolic heuristic search for factored markov decision processes. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)*.
- Feng, Z.; Hansen, E. A.; and Zilberstein, S. 2003. Symbolic generalization for on-line planning. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence*.
- Hansen, E., and Zilberstein, S. 2001. LAO\*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129:35–62.
- Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*, 279–288.
- St-Aubin, R.; Hoey, J.; and Boutilier, C. 2000. APRICODD: Approximate policy construction using decision diagrams. In *Proceedings of NIPS-2000*.

# NMRDPP: Decision-Theoretic Planning with Control Knowledge.

Charles Gretton, David Price, and Sylvie Thiébaux

Computer Sciences Laboratory  
The Australian National University  
Canberra, ACT, Australia

{charlesg,davidp,thiebaux}@csl.anu.edu.au

## Abstract

We discuss NMRDPP, a system for solving decision processes with non-Markovian reward. More specifically, target decision processes exhibit Markovian dynamics and rewarding behaviours are modelled as state trajectories specified in a linear temporal logic. In addition to implementing structured, tabular and online MDP solution algorithms, NMRDPP can exploit domain specific control knowledge. State trajectories which violate the users knowledge/intuition regarding useful dynamics can be pruned from consideration by the MDP solution algorithm. Thus, in addition to facilitating concise specification of complex reward structures, NMRDPP can be used to greatly speed up policy computation for propositional MDPs. To our knowledge, NMRDPP is the only implementation of solution algorithms designed to solve decision processes with non-Markovian rewards.

## Introduction

NMRDPP (Gretton *et al.* 2003) (non-Markovian Reward Decision Process Planner), is a general purpose planner for non-Markovian reward<sup>1</sup> (and hence also Markovian) propositional decision processes. Target decision processes are usually stochastic, exhibiting Markovian dynamics. The reward is modelled as a set of state trajectories, called behaviours, specified in a linear temporal logic. NMRDPP was originally developed in order to carry out an experimental evaluation of approaches for solving decision processes with non-Markovian reward. Implemented in C++, NMRDPP supports a range of experimental algorithms and frameworks for solving NMRDPs. It is suited to participation in IPPC'04 as it facilitates planning in completely observable stochastic domains. NMRDPP is the first of its kind; previously no approaches to solving NMRDP had been fully implemented, and there was no work presenting any experimental results.

There have been two proposals regarding languages suitable for expressing rewarding behaviours. These include PLTL (Bacchus *et al.* 1996) a linear temporal logic of the past and \$FLTL (Thiébaux *et al.* 2002) a linear temporal logic of the future with reward. In either case, NMRDPP translates NMRDPs into corresponding equivalent MDPs (XMDPs) which incorporate temporal variables capturing sufficient history to make the reward of the expanded

<sup>1</sup>For our purposes reward can be negative, thus we don't distinguish between reward and cost.

process Markovian<sup>2</sup>. Available translation procedures are unique and not particularly straightforward (Bacchus *et al.* 1996; Bacchus *et al.* 1997; Thiébaux *et al.* 2002). NMRDP solution algorithms differ in their representations of domain dynamics, the XMDP and in the class, structured or non-structured, of MDP solution methods to which they are tied. NMRDPP can solve target decision problems online (during translation) using LAO\* heuristic search techniques (Hansen and Zilberstein 2001). Alternatively, the complete XMDP can be generated and passed to classical structured or tabular policy computation algorithms such as SPUD (Boutilier *et al.* 1995; Hoey *et al.* 1999) or policy/value iteration (Howard 1960) respectively.

Using the same mechanisms devised for non-Markovian reward, state trajectories which violate the users knowledge/intuition regarding useful dynamics can be pruned from consideration by the MDP solution algorithm. The specification of a set of such state sequences is called *control knowledge*, and has been used to great effect by the deterministic planning community (Bacchus and Kabanza 2000). Thus, although there is no advantage to be gleaned from concise specification of complex non-Markovian reward during the competition, NMRDPP can exploit control knowledge to greatly speed up policy computation given propositional MDPs. By pruning states which violate specific behaviours, we can mitigate the effect of Bellman's so called curse of dimensionality.

In the remainder of this document, we shall present an overview of MDPs and NMRDPs and discuss their differences. We shall briefly discuss the logics that have been adopted to model reward and control knowledge, focusing in particular on \$FLTL. We shall provide some examples of using \$FLTL to specify control knowledge for a stochastic blocks-world domain. We shall conclude by summarising how we intend to compete using NMRDPP in the IPPC'04.

## MDPs and NMRDPs

Problem domains which participants shall consider during the *main* and *learning* IPPC'04 tracks, although specified in PPDDL1.0 (Younes and Littman 2004), can be modelled using the MDP formalism. Indeed, decision theoretic

<sup>2</sup>There is a mapping from XMDP states to the reals.

planning problems are typically modelled as propositional MDPs such that domain states  $S$  are characterised by propositions, numeric reward is allocated to propositions/states according to their associated desirability and the dynamics of the system is given by actions  $A$ . We typically write  $A(s)$  to denote actions applicable at state  $s$ . A solution algorithm, provided with start states  $S_0 \subseteq S$ , generates a stationary policy  $\pi : S \rightarrow A$  (mapping from states to actions) which adherence to during system execution results in optimal behaviour over a discounted infinite horizon.

The standard MDP formulation is state based, comprising a *finite* set of states  $S$  and actions  $A$ . Actions induce stochastic state transitions, where  $s, t \in S$ ,  $a \in A$  and  $Pr(s, a, t)$  gives the probability of a transition from state  $s$  to  $t$  given action  $a$  is executed at state  $s$ . Also present is a real-valued reward function  $R : S \rightarrow \mathbb{R}$ . The value of a stationary policy  $\pi$  at a state  $s_0 \in S_0$ ,  $V(\pi)$ , is given by Equation 1.

$$V(\pi) = \lim_{n \rightarrow \infty} \mathbb{E} \left[ \sum_{i=0}^n \beta^i R(\Gamma_i) \mid \pi, \Gamma_0 \in S_0 \right] \quad (1)$$

Here  $\beta$  is a discount factor usually close to 1 and  $\Gamma \in S^*$  is a finite sequence of states where  $\Gamma_i$  is the  $i$ 'th state in  $\Gamma$ . We consider a policy  $\pi^*$  optimal if, for all  $\pi$ , we have that  $V(\pi^*) \geq V(\pi)$ .

The formulation for NMRDPs is identical up to the reward function whose domain is extended to  $S^*$ , e.g.  $R : S^* \rightarrow \mathbb{R}$ . Here,  $\Gamma(i)$  is the  $i$  length subsequence of  $\Gamma$  starting at  $\Gamma_0$ . As before, the value of  $\pi$ , which we seek to maximise, is the expectation of the discounted cumulative reward over an infinite horizon:

$$V(\pi) = \lim_{n \rightarrow \infty} \mathbb{E} \left[ \sum_{i=0}^n \beta^i R(\Gamma(i)) \mid \pi, \Gamma_0 \in S_0 \right] \quad (2)$$

As introduced, NMRDP solution methods facilitate generation of an optimal policy by first expanding the NMRDP into an XMDP, and then applying either traditional or structured MDP solution algorithms to the resulting construct.

## Reward Specification and Control Knowledge

NMRDPP supports the use of two linear temporal logics in the specification of rewarding state trajectories and control knowledge. These are PLTL (Bacchus *et al.* 1996) and \$FLTL (Thiébaux *et al.* 2002). The logic PLTL includes the modalities  $\odot$  (previously),  $\mathbf{S}$  (since),  $\Diamond f \equiv \top \mathbf{S} f$  (once) and  $\Box f \equiv \neg \Diamond \neg f$  (always in the past) while \$FLTL includes  $\bigcirc \phi$  (next),  $\mathbf{U}$  (weak until),  $\Box \phi \equiv \phi \mathbf{U} \perp$  (always), and a propositional constant  $\$$  (receive reward now)<sup>3</sup>.

A translation from an NMRDP into a corresponding MDP is based on the fact that a PLTL, resp. \$FLTL, wff  $\phi$  can be regressed (Bacchus and Kabanza 2000), resp. progressed, to a formula which identifies what must hold in the past, resp. future, for  $\phi$  to hold in a current state. Given this progression/regression operator, methods annotate grounded states

<sup>3</sup>See the respective papers for a comprehensive summary of the two logics.

to form expanded states with formulae (temporal variables) which are sufficient to determine the reward allocation at any state reachable from  $S_0$ . Methods are characterised by the properties of the XMDP which they generate. Given a PLTL reward specification NMRDPP can attempt to generate the minimal MDP required to allocate reward given specified behaviours. Using the language \$FLTL, NMRDPP is able to produce a blind minimal XMDP online. Intuitively, a blind minimal XMDP is the smallest MDP achievable by online translation.

As a derivative of FLTL, \$FLTL is particularly suitable for expressing domain specific control knowledge (Bacchus and Kabanza 2000) which is useful in the context of online solution algorithms. That is, a decision process can be modified by excluding from it sequences of states which violate a control hypotheses expressed in \$FLTL.

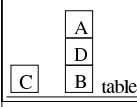
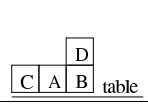
## Blocks-World with \$FLTL Control Knowledge

We have that a stochastic version of blocks-world will feature in the learning track at IPPC'04. Using the language of PPDDL1.0, BW has `block` and `table` types, predicates `holding : block` and `On : (block  $\times$  T)`, and two action symbols `pick-up : (block  $\times$  T)` and `put-down : (block  $\times$  T)`. A BW state comprises two ground sets of predicate symbols, those which characterise the current and goal states. Pictorially, a BW state appears as follows:

" $\forall b. \neg \text{holding}(b)$ "

**Goal:** `On(D, B)`, `On(B, table)`,  
..., `On(C, table)`

**Conf:** `On(A, D)`, `On(D, B)`,  
..., `On(C, table)`

	
Current configuration	Goal configuration

We have that the `pick-up(a, b)` action is only executable if `table = a  $\vee$   $\neg$ block(c).On(c, a)` (i.e.  $a$  is clear). Similarly `put-down(a, b)` is only available if `holding(a)` and  $b$  is clear. Assuming precondition satisfaction, `pick-up(a)` either causes `block a` to be held, or possibly fall on the `table`. The action `put-down(a, b)` either drops  $a$  on the `table` or places it on the second argument object  $b$ .

This stochastic BW isn't particularly different from its deterministic relatives. Thus, we can appeal to near optimal planning strategies such as US and GN1 (Slaney and Thiébaux 2001) in developing control knowledge. For the purposes of this presentation we introduce the predicate `InPosition(a, b)` for  $a$  and  $b$  of type `block`. `InPosition(a, b)` is false if `On(a, b)` is not an element of the goal configuration, and otherwise true when  $a$  and  $b$  are in their goal position. Notice that this is a derived predicate, i.e. given our example state, `InPosition(D, B)  $\equiv$  On(D, B)  $\wedge$  On(B, table)`. Thus, the following control knowledge is expressible without change to the competition specification.

The first piece of control knowledge that we consider prevents NMRDPP from disturbing towers of blocks which satisfy the goal. For each `On( $b_i, b_j$ )`, a control sentence of the following form is sufficient:

$$\Box( (\text{On}(b_i, b_j) \wedge \text{InPosition}(b_i, b_j)) \rightarrow (\neg \text{holding}(b_i)))$$

By pruning from a BW domain states which violate the above sentence, NMRDPP will not consider policies which disturb blocks that are in their goal position. We can further prune the range of policies which NMRDPP shall consider by noticing that if  $\text{On}(b_i, b_j)$  is false where  $b_j$  is a block, and after two action invocations  $\text{On}(b_i, b_j)$ , then this is only valuable where  $\text{InPosition}(b_i, b_j)$ . This knowledge is expressed in \$FCTL as follows.

$$\Box( (\neg \text{On}(b_i, b_j) \wedge \bigcirc \bigcirc (\text{On}(b_i, b_j))) \rightarrow (\bigcirc \bigcirc \text{InPosition}(b_i, b_j)))$$

Annotating BW problems with the above control knowledge greatly increases the situations in which NMRDPP is competitive. Because progression of \$FCTL formulae is linear time in the formula length, it is important that we avoid crippling NMRDPP by providing too much, mostly redundant or too complex knowledge. Furthermore, we must ensure that knowledge generation for competition domains is practical.

## Participation

NMRDPP, provides an implementation of several solution approaches in a common framework, within a single system, and with a common input language. The framework includes a highly interactive command line interface which allows the user to exert fine control over the planning process. The input language enables specification of actions, initial states, rewards, and control-knowledge. Initial states are specified as part of the control knowledge or as explicit assignments to propositions. Of interest to us here is the format for the action specification, which is essentially the same as in the SPUDD system (Hoey *et al.* 1999). In particular, the precondition (BDD), reward and probabilistic effects for each action are specified by a collection of decision trees, including one for each domain proposition which the action effects. When the input is parsed, the action specification trees are converted into ADDs by the CUDD package (Somenzi 2001).

The input language in which competition domains are specified is PPDDL1.0. We will be able to accommodate this in one of two ways. 1) Because both NMRDPP and the competition software code are implemented in C++, we can directly take advantage of competition code which facilitates exploration of the explicit propositionalised state space. In this case we restrict NMRDPP to state-based solution algorithms. 2) We can translate PPDDL1.0 problem specifications into the NMRDPP input language. This is an enticing option because the competition code contains functionality implemented by Håkan L. S. Younes which encodes grounded problem actions as ADDs. There is insufficient space for us to include the details here, however such grounded action ADDs can be converted into action descriptions in the NMRDPP input format. In essence, such a process extracts diachronic and synchronic dependencies between propositions in an action's conditional probabilistic model in order to construct decision trees/ADDs en-

coding action preconditions, reward and effects on individual propositions. Where we generate NMRDPP input from PPDDL1.0, the action specifications are not concise<sup>4</sup> as information regarding pre/post-action-variable dependencies is lost in translation. The advantage however, is that we do not restrict ourselves to a subset of solution algorithms supported by NMRDPP.

We intend to enter NMRDPP in both the *main* and *learning* tracks of IPPC'04. In the *main* track we do not expect much from NMRDPP as it is laden with some overhead due to support for non-Markovian reward. For the *Learning Track*, we shall develop "hand coded" control knowledge specific to each competition domain in order to make NMRDPP competitive. Although this does not position us well to compete with first-order learners which are not restricted to a propositional domain model, we hope to be competitive in small to medium domain instances. At the time of writing, it was not clear which of the structured, tabular and online algorithms NMRDPP supports, we shall use in the competition.

## References

- F. Bacchus and F. Kabanza. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2), 2000.
- F. Bacchus, C. Boutilier, and A. Grove. Rewarding behaviors. In *Proc. AAAI-96*, pages 1160–1167, 1996.
- F. Bacchus, C. Boutilier, and A. Grove. Structured solution methods for non-markovian decision processes. In *Proc. AAAI-97*, pages 112–117, 1997.
- C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *Proc. IJCAI-95*, pages 1104–1111, 1995.
- Charles Gretton, David Price, and Sylvie Thiébaux. Implementation and comparison of solution methods for decision processes with non-markovian rewards. In *Proc. UAI-03*, 2003.
- E. Hansen and S. Zilberstein. LAO\*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129:35–62, 2001.
- J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: stochastic planning using decision diagrams. In *Proc. UAI-99*, 1999. SPUDD is available from <http://www.cs.ubc.ca/spider/staubin/Spudd/>.
- R.A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.
- J. Slaney and S. Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125:119–153, 2001.
- F. Somenzi. CUDD: CU Decision Diagram Package. Available from <ftp://vlsi.colorado.edu/pub/>, 2001.
- S. Thiébaux, F. Kabanza, and J. Slaney. Anytime state-based solution methods for decision processes with non-markovian rewards. In *Proc. UAI-02*, pages 501–510, 2002.
- H. Younes and M. Littman. PPDDL1.0: An extension to PDDL for Expressing Planning Domains with Probabilistic Effects, 2004. <http://www.cs.cmu.edu/lorens/papers/ppddl.pdf>.

<sup>4</sup>We find that for large domains, i.e. BW with 10 > blocks for example, a translation into NMRDPP input format is impractical.

# FCPlanner: A Planning Strategy for First-Order MDPs

**Eldar Karabaev**

Institute for Theoretical Computer Science  
Dresden University of Technology  
Dresden, Germany  
karabaev@tcs.inf.tu-dresden.de

**Olga Skvortsova\***

Institute for Artificial Intelligence  
Dresden University of Technology  
Dresden, Germany  
skvortsova@inf.tu-dresden.de

## Introduction

FCPLANNER (Fluent Calculus Planner) is a planning system that is based on the first-order value iteration algorithm (FOVIA) (Großmann, Hölldobler, & Skvortsova 2002) for solving first-order MDPs. Following the idea of symbolic dynamic programming (SDP) within the Situation Calculus by Boutilier and colleagues (Boutilier, Reiter, & Price 2001), FOVIA addresses the well-known scalability problem of the classical dynamic programming algorithms by employing the abstraction technique, i.e., a state space is divided into clusters, called *abstract states*, and the value functions are computed for them thereafter. The dynamics of an MDP is formalized in the probabilistic Fluent Calculus (pFC) that allows for introducing stochastic actions. Our approach constructs a first-order representation of value functions and policies by exploiting the logical structure of the MDP. Thus, FOVIA can be seen as a symbolic (logical) counterpart of classical value iteration algorithm (Bellman 1957).

## Abstract States

We formalize abstract states symbolically, within the Fluent Calculus (FC) (Hölldobler & Schneeberger 1990). Fluent Calculus, much like Situation Calculus, is a logical approach to modelling dynamically changing systems based on first-order logic. One could indeed argue that Fluent Calculus and Situation Calculus have very much in common. But the latter has the following disadvantage: Knowledge of the current state is represented indirectly via the initial conditions and the actions which the agent has performed up to a point. As a consequence, each time a condition is evaluated in an agent program, the entire history of actions is involved in the computation. This requires ever increasing computational effort as the agent proceeds, so that this concept does not scale up well to long-term agent control (Thielscher 2004). Fluent Calculus overcomes the aforementioned unfolding problem by providing the crucial concept of an explicit state representation. The information on what is true in the current state of the world is effortlessly extracted from the state description without tracing back to the initial state. Therefore we have opted for Fluent Calculus as logical formalism

underlying our automated symbolic dynamic programming approach.

In FC, functions whose values vary from state to state are called *fluents* and are denoted by function symbols. For example, the fluent  $on(X, table)$  denotes the presence of a block  $X$  on the table. A *state* is a multiset of fluents represented as a term, called *fluent term*, using a constant 1 denoting the empty multiset and a binary function symbol  $\circ$  denoting multiset union that is associative, commutative and admits unit element. For example, a state in which the block  $a$  is on the block  $b$  and  $b$  is on the table is specified by  $on(a, b) \circ on(b, table)$ . Constants are denoted by small letters, variables by capital ones and substitutions by  $\theta$  or  $\sigma$ .

*Abstract states* are characterized by means of conditions that must hold in each ground instance thereof and, thus, they represent sets of real-world states. Informally, abstract states can be specified by stating that particular fluent terms do or do not hold. We refer to such abstract states as *CN-states*, where  $C$  stands for conjunction and  $N$  for negation, respectively.

Formally, let  $\mathcal{L}$  be a set of fluent terms. A *CN-state* is a pair  $(P, \mathcal{N})$ , where  $P \in \mathcal{L}$ ,  $\mathcal{N} \in 2^{\mathcal{L}}$ . Let  $\cdot^M$  be a mapping from fluent terms to multisets of fluents, which can be formally defined as follows:  $1^M = \{\}$  or  $F^M = \{F\}$ , if  $F$  is a fluent, or  $(F \circ G)^M = F^M \dot{\cup} G^M$ , where  $F, G$  are fluent terms and  $\dot{\cup}$  is a multiset union. Let  $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$  be an interpretation, whose domain  $\Delta$  is the set of all finite multisets of ground fluents and every *CN-state*  $Z = (P, \mathcal{N})$  is mapped onto

$$Z^{\mathcal{I}} = \{d \in \Delta \mid \exists \theta. (P\theta)^M \dot{\subseteq} d \wedge \forall N \in \mathcal{N}. \forall \sigma. ((N\theta)\sigma)^M \not\subseteq d\},$$

where  $\dot{\subseteq}$  is a submultiset relation.

In other words, the  $P$ -part of a state  $Z$  describes properties that a real-world state should satisfy, whereas  $\mathcal{N}$ -part specifies the properties that are not allowed to fulfil. For example, the *CN-state*  $Z = (on(X, table) \circ red(X), \{on(Y, X)\})$  represents all states in which there exists a red object that is on the table and clear, viz., none of other objects covers it.

Thus, the real-world state

$$z = \{on(a, table), red(a), on(b, table), green(b)\}$$

\*Supported by the research training group GRK 334/3 (DFG). Corresponding author.

is specified by  $Z$ . Whereas,

$$z' = \{on(a, table), red(a), on(b, a)\}$$

is not.

Intuitively, *CN-states* can be represented as first-order formulae. The above-given *CN-state*  $Z$  corresponds to the following formula:

$$\exists X.on(X, table) \wedge red(X) \wedge \forall Y.\neg on(Y, X) .$$

Please note that *CN-states* should be thought of as incomplete state descriptions, i.e., the properties that are not listed in either  $P$ - or  $\mathcal{N}$ -part can hold or not.

## Stochastic Actions

The technique for introducing stochastic actions within the probabilistic Fluent Calculus is to decompose a stochastic action into deterministic primitives under nature's control, referred to as *nature's choices*. We use a relation symbol *choice*/2 to model nature's choice. Consider the action *putdown*( $T, B$ ) of putting a block  $T$  down onto a block  $B$  from the blockworld scenario:

$$choice(putdown(T, B), A) \leftrightarrow (A = putdown_1(T, B) \vee A = putdown_2(T, B)),$$

where *putdown*<sub>1</sub>( $T, B$ ) and *putdown*<sub>2</sub>( $T, B$ ) define two nature's choices for action *putdown*( $T, B$ ). The nature's choice *putdown*<sub>1</sub>( $T, B$ ) states the successful putting of the block  $T$  down onto  $B$ . Whereas, *putdown*<sub>2</sub>( $T, B$ ) defines the failure execution of the *putdown*-action which results in the block  $T$  falling down on the table.

For each of nature's choices  $a_j(\bar{X})$  associated with an action  $a(\bar{X})$  with parameters  $\bar{X}$  we define the probability  $prob(a_j(\bar{X}), a(\bar{X}), Z)$ . It denotes the probability with which one of nature's choices  $a_j(\bar{X})$  is chosen in a *CN-state*  $Z$ . For example,

$$prob(putdown_1(T, B), putdown(T, B), Z) = .7$$

states that the probability for the successful execution of the *putdown* action in  $Z$  is .7.

FOVIA is an iterative approximation algorithm for constructing optimal policies. The difference to classical case is that it produces a first-order representation of optimal policies by utilizing the logical structure of MDP. The algorithm itself can be found in (Großmann, Hölldobler, & Skvortsova 2002).

## Preprocessing

In order to convert a PPDDL goal description into a goal state space that is used as an input of our FOVIA algorithm, we have designed a procedure for translating first-order formulae into a set of *CN-states*.

Since a state space is considered as a disjunction of *CN-states*, we first convert a FO formula into DNF. We start with pushing all quantifiers in front of the formula and convert the quantifier-free part into DNF thereof. In order to check whether a disjunct can be directly converted into a

*CN-state*, we have to examine its variables. If a disjunct contains no 'bad' variables then it can be directly converted into a respective *CN-state*. Otherwise, the formula itself needs an additional treatment.

The procedure of marking variables as 'bad' works as follows: If a variable occurring within a positive literal is bounded universally then it is marked as 'bad'. Intuitively, based on the semantics of *CN-states*, the variables that occur in the  $P$ -part of a *CN-state* are considered existentially bounded. Each 'bad' variable is eliminated via groundization.

For example, in the following formula

$$\forall X.\exists Y.red(X) \wedge blue(Y)$$

the variable  $X$  will be marked as 'bad'.

Assume that we have only two blocks  $a$  and  $b$  in the domain. After eliminating  $X$  (and slight simplification), we obtain:

$$red(a) \wedge red(b) \wedge \exists Y.blue(Y) .$$

The variable  $Y$  will not be marked as 'bad', hence, it will not be grounded. Similarly, the negative literals are checked for 'bad' variables. The same technique for eliminating 'bad' variables is applied for action descriptions.

Although our approach relies on partial groundization of state and action descriptions, there are domains, e.g., colored blockworld, where most variables are marked as 'good', and hence, need not be grounded.

## Regression of Abstract States

The classical as well as first-order value iteration algorithms are intimately related to regression of states. The crucial difference of the symbolic value iteration is that the regression is performed on the abstract states instead of the single states themselves.

Given a *CN-state*  $Z$  and an action description  $A$ , our regression procedure produces the set of all possible predecessor *CN-states*  $Z_i$  such that  $Z$  is reachable from each of  $Z_i$  by executing  $A$ . In FOVIA, actions are specified by preconditions that are represented as *CN-states* and STRIPS style effects  $Q^+$  and  $Q^-$ .

We now illustrate the regression procedure with an example from the blockworld scenario. Here, we present one regression step through action *putdown*( $Top, Bottom$ ) that has two nature's choices, given below:

$$\begin{aligned} &putdown_1(Top, Bottom) \\ &\text{Pre} : (holding(Top), \{on(X, Bottom)\}) \\ &\text{Eff} : Q^+ = on(Top, Bottom) \\ &\quad Q^- = holding(Top) \\ &putdown_2(Top, Bottom) \\ &\text{Pre} : (holding(Top), \{on(X, Bottom)\}) \\ &\text{Eff} : Q^+ = on(Top, table) \\ &\quad Q^- = holding(Top) . \end{aligned}$$

The regression of the *CN-state*  $Z$ :

$$Z = (on(B_0, B_1) \circ on(B_1, table) \circ on(B_2, table), \emptyset)$$

yields the following predecessor states  $Z_i$ :

$$\begin{aligned} Z_1 &= (\text{holding}(B_2) \circ \text{on}(B_0, B_1) \circ \text{on}(B_1, \text{table}), \emptyset) \\ Z_2 &= (\text{holding}(B_2) \circ \text{on}(B_0, B_1) \circ \text{on}(B_1, \text{table}) \circ \\ &\quad \text{on}(B_3, \text{table}), \{\text{on}(B_4, B_3)\}) \\ Z_3 &= (\text{holding}(B_0) \circ \text{on}(B_1, \text{table}) \circ \text{on}(B_2, \text{table}), \\ &\quad \{\text{on}(B_3, B_1)\}) , \end{aligned}$$

where  $Z_1$  represents all real-world states, where a gripper holds a block  $B_2$ , a block  $B_0$  is on  $B_1$  and  $B_1$  is on the table;  $Z_2$  asserts the same information as  $Z_1$  and additionally states that some block  $B_3$  is on the table and there is no such block  $B_4$  that is on  $B_3$ ; and  $Z_3$  is interpreted as the set of all real-world states, where a gripper holds a block  $B_0$ , blocks  $B_1$  and  $B_2$  are on the table, and there is no such block  $B_3$  that is on  $B_1$ .

The regression procedure can be outlined as follows. We first check whether the  $Q^-$  effects and the  $P$ -part of a  $CN$ -state  $Z$  are consistent wrt. each other. If the answer is no, then the regression procedure stops delivering the empty set of predecessor  $CN$ -states. Otherwise, a predecessor state is constructed as follows: The  $Q^+$  effects are subtracted from the  $P$ -part of the  $CN$ -state  $Z$  and the result is joined with the  $P$ -part of the action preconditions forming the  $P$ -part of a predecessor  $CN$ -state. Analogously, the  $N$ -part of a predecessor  $CN$ -state is built by subtracting the  $Q^-$  effects from the  $N$ -part of  $Z$  and joining the result with the  $N$ -part of the action preconditions. If the resulting predecessor state is consistent then it is added to the set of the  $Z$ 's predecessor states. We describe the consistency check in more detail in the section on optimizations.

The operations over fluent terms and sets of fluent terms, e.g., aforementioned subtraction and union, are based on solving the submultiset matching problem that usually has multiple solutions (Große *et al.* 1992). This implies that the regression procedure may deliver multiple predecessor states. Recalling our running example, both  $CN$ -states  $Z_1$  and  $Z_3$  were obtained as a result of the regression of  $Z$  through a single nature's choice  $\text{putdown}_1$ .

### Some Optimizations

In general, a state description may contain two kinds of inconsistencies. The inconsistency of the first kind takes place when some element of the  $N$ -part contradicts with the  $P$ -part. For example, in a state description  $(\text{red}(a), \{\text{red}(X)\})$  the  $P$ -part asserts that the block  $a$  is red, whereas the  $N$ -part prohibits any block  $X$  of being red. In this case, the consistency test will include a simple syntactic check.

The second kind of inconsistencies is referred to as domain-dependent. For example, the state description  $(\text{empty} \circ \text{holding}(a), \emptyset)$  is formally consistent (wrt. the previous kind of inconsistency). And only after having learned that the domain contains a single gripper, this  $CN$ -state is turned to be inconsistent. In this case, the consistency test uses additional domain axioms which, e.g., state that the combination of fluents  $\text{empty}$  and  $\text{holding}(X)$  is forbidden.

The state space that represents a value function after some iteration step of FOVIA algorithm may contain redundancies. For example, consider a state space that consists of two abstract states  $Z_1 = (\text{holding}(a), \emptyset)$  and  $Z_2 =$

$(\text{holding}(X), \emptyset)$  that are both assigned the same value, say, of 10. The  $CN$ -state  $Z_1$  represents the set of all real-world states that do satisfy the fact  $\text{holding}(a)$ . At the same time, the  $CN$ -state  $Z_2$  describes all real-world states represented by  $Z_1$  plus additional states, where  $X$  is instantiated by a constant different from  $a$ . Since the values associated with  $Z_1$  and  $Z_2$  are the same,  $Z_1$  can be painlessly removed without loss of information. In FCPLANNER, we employ the automated normalization procedure that, given a state space, delivers an equivalent one that contains no redundancies (Skvortsova 2003). The technique employs the notion of a subsumption relation that enables to determine which states are redundant and can be removed from the state space therefore.

### References

- Bellman, R. E. 1957. *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press.
- Boutilier, C.; Reiter, R.; and Price, B. 2001. Symbolic Dynamic Programming for First-Order MDPs. In Nebel, B., ed., *Proceedings of the Seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, 690–700. Morgan Kaufmann.
- Große, G.; Hölldobler, S.; Schneeberger, J.; Sigmund, U.; and Thielscher, M. 1992. Equational logic programming, actions, and change. 177–191. MIT Press.
- Großmann, A.; Hölldobler, S.; and Skvortsova, O. 2002. Symbolic Dynamic Programming within the Fluent Calculus. In Ishii, N., ed., *Proceedings of the IASTED International Conference on Artificial and Computational Intelligence*, 378–383. Tokyo, Japan: ACTA Press.
- Hölldobler, S., and Schneeberger, J. 1990. A new deductive approach to planning. *New Generation Computing* 8:225–244.
- Skvortsova, O. 2003. Towards Automated Symbolic Dynamic Programming. Master's thesis, TU Dresden.
- Thielscher, M. 2004. FLUX: A logic programming method for reasoning agents. *Theory and practice of Logic Programming*.

# Probapop: Probabilistic Partial-Order Planning

Nilufer Onder      Garrett C. Whelan      Li Li

Department of Computer Science  
Michigan Technological University  
1400 Townsend Drive  
Houghton, MI 49931  
{nilufer,gcwhelan,lili}@mtu.edu

## Abstract

We describe Probapop, a partial-order probabilistic planning system. Probapop is a blind (conformant) planner that finds plans for domains involving probabilistic actions but no observability. The Probapop implementation is based on Vhpop, a partial-order deterministic planner written in C++. The Probapop algorithm uses plan graph based heuristics for selecting a plan from the search queue, and probabilistic assessment heuristics for selecting a condition whose probability can be increased.

## Introduction

Probapop<sup>1</sup> is a *conformant probabilistic planner* (term used in (Hyafil & Bacchus 2003)). In this paradigm, the actions and the initial state can be *probabilistic*, i.e., they can have several possible outcomes annotated by a probability of occurrence. In addition, the planning problem is *conformant* i.e., the agent cannot observe the environment. The objective is to find a minimal sequence of steps that will take an agent from an initial set of states to a specified goal state within a specified threshold probability. Note that while the assumption of blind agents is not true in general, it is useful to incorporate conformant planning methods because sensing might be expensive, not reliable, or not available. We leave contingency planning, e.g., (Majercik & Littman 1999; Onder & Pollack 1999; Hansen & Feng 2000; Karlsson 2001) and other paradigms that assume non-probabilistic effects, e.g., (Ferraris & Giunchiglia 2000; Bertoli, Cimatti, & Roveri 2001) outside the current implementation of Probapop.

Our work is motivated by the incentive to have partial-order planning as a viable option for conformant probabilistic planning. The primary reason is that partial-order planners have worked very well with lifted actions which are useful in coding large domains

in a compact way. Second, due to its least commitment strategy in step ordering, partial-order planning (POP) produces plans that are highly parallelizable. Third, planners that can handle rich temporal constraints have been based on POP algorithms (Smith, Frank, & Jonsson 2000).

Our basic approach is to form base plans by using deterministic partial-order planning techniques, and then to estimate the best way to improve these plans. Recently Repop (Nguyen & Kambhampati 2001) and Vhpop (Younes & Simmons 2002) planners have demonstrated that the very heuristics that speed up non-partial-order planners can be used to scale up partial-order planning. We show that these distance based heuristics (McDermott 1999; Bonet & Geffner 1999) as implemented using “relaxed” plan graphs can be employed in probabilistic domains. These, coupled with selective plan improvement heuristics result in significant improvement. As a result, *Probapop* enjoys the soundness, completeness, and least-commitment properties of partial-order planning and makes partial-order planning feasible in probabilistic domains.

## Probapop and Partial-Order Planning

For partial-order probabilistic planning, we implemented the Buridan (Kushmerick, Hanks, & Weld 1995) probabilistic planning algorithm on top of Vhpop (Younes & Simmons 2002), a recent partial-order planner. A partially ordered plan  $\pi$  is a 6-tuple,  $\langle \text{STEPS}, \text{ORD}, \text{BIND}, \text{LINKS}, \text{OPEN}, \text{UNSAFE} \rangle$ , representing sets of ground actions, ordering constraints, binding constraints, causal links, open conditions, and unsafe links, respectively. An ordering constraint  $S_i \prec S_j$  represents the fact that step  $S_i$  precedes  $S_j$ . A *causal link* is a triple  $\langle S_i, p, S_j \rangle$ , where  $S_i$  is the *producer*,  $S_j$  is the consumer and  $p$  represents the condition supported. An *open condition* is a pair  $\langle p, S \rangle$ , where,  $p$  is a condition needed by step  $S$ . A causal link  $\langle S_i, p, S_j \rangle$  is *unsafe* if the plan contains a *threatening* step  $S_k$  such that  $S_k$  has  $\bar{p}$  among its effects, and

<sup>1</sup>This work has been supported by a Research Excellence Fund grant from Michigan Technological University.



$S_k$  may intervene between  $S_i$  and  $S_j$ . Open conditions and unsafe links are collectively referred to as *flaws*. A *planning problem* is a triple  $(I, G, t)$ , where, the initial state  $I$  is a probability distribution over states,  $G$  is a set of literals that must be true at the end of execution, and  $t$  is a probability threshold. The planner must find a plan that takes the agent from  $I$  to  $G$  with a probability  $\geq t$ . If several plans have the same probability of success, then the one with the least number of steps is preferred.

The Probapop algorithm shown in Fig. 1 first constructs an initial plan by forming  $I$  and  $G$  into initial and goal steps, and then refines the plans in the search queue until it finds a solution plan that meets or exceeds the probability threshold. Plan refinement operations involve repairing flaws. An open condition can be closed by adding a new step from the domain theory, or reusing a step already in the plan. An unsafe link is handled by the *promotion*, *demotion*, or *separation* (lifted actions are used) operations, or by *confrontation* (Penberthy & Weld 1992) which involves commitment to non-threatening effects.

```

function PROBAPOP (initial, goal, t)
returns a solution plan, or failure
  plans  $\leftarrow$  MAKE-MINIMAL-PLAN(initial, goal)
  loop do
    if plans is empty then return failure
    plan  $\leftarrow$  REMOVE-FRONT(plans)
    if SOLUTION?(plan, t) then return plan
    plans  $\leftarrow$  MERGE(plans, REFINES-PLAN(plan))
  end

function REFINES-PLAN (plan)
returns a set of plans (possibly null)
  if FLAWS(plan) is empty then
    plan  $\leftarrow$  REOPEN-CONDITIONS(plan)
  flaw  $\leftarrow$  SELECT-FLAW(plan)
  if flaw is an open condition then choose:
    return REUSE-STEP(plan, flaw)
    return ADD-NEW-STEP(plan, flaw)
  if flaw is a threat then choose:
    return DEMOTION(plan, flaw)
    return PROMOTION(plan, flaw)
    return SEPARATION(plan, flaw)
    return CONFRONTATION(plan, flaw)

```

Figure 1: The probabilistic POP algorithm.

The search is conducted using an A\* algorithm guided by a *ranking function*  $f$ . As usual for a plan  $\pi$ ,  $f(\pi) = g(\pi) + h(\pi)$ , where  $g(\pi)$  is the cost of the plan, and  $h(\pi)$  is the estimated cost of completing it. In Probapop,  $g$  reflects the number of steps in a plan,  $h$  represents the estimated number of steps to complete a plan. Both are weighted by the probability of success of the overall plan. The ranking function is used at the

MERGE step to order the plans in the search queue such that the plan that ranks best is at the beginning of the queue. We term a plan for which  $\text{OPEN} = \text{UNSAFE} = \emptyset$  as a *quasi-complete* plan. A quasi-complete plan is not a solution if it does not meet the probability threshold. Probapop can be viewed as first choosing a plan to improve using the ranking function, then choosing a way to improve the plan, and finally choosing a way to implement the improvement. These phases do not have to follow strictly or work on the same plan. After the successors of a plan are generated, the ranking function might gear the search toward other plans in the search queue. In the next section, we describe the heuristics used.

## Distance Based Ranking and Selective Reopening in Probapop

The Vhpop deterministic partial order-planner described in (Younes & Simmons 2002) implements the *ADD heuristic* to provide an estimate of the total number of new actions needed to close all the open conditions. Before starting to search, the planner builds a planning graph (Blum & Furst 1997) which has the literals in the initial state in its first level, and continues to expand it until it reaches a level where all the goal literals are present. Vhpop's *ADD* heuristic achieves good performance by computing the step cost of the open conditions from the planning graph, i.e.,  $h_{add}(\pi) = h_{add}(\text{OPEN}(\pi))$ . The cost of achieving a literal  $q$  is the level of the first action that achieves  $q$ :  $h_{add}(q) = \min_{a \in GA(q)} h_{add}(a)$  if  $GA(q) \neq \emptyset$ , where  $GA(q)$  is an action that has an effect  $q$ . Note that  $h_{add}(q)$  is 0 if  $q$  holds initially, and is  $\infty$  if  $q$  never holds. The *level* of an action is the first level its preconditions become true:  $h_{add}(a) = 1 + h_{add}(\text{PREC}(a))$ .

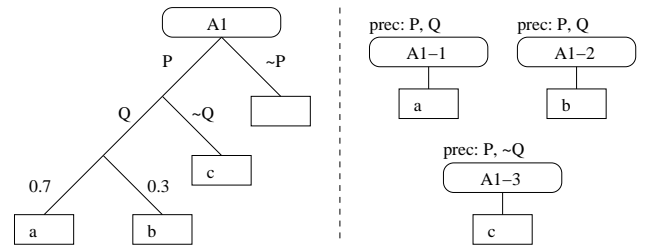


Figure 2: Probabilistic action A1 is split into deterministic actions A1-1, A1-2, and A1-3.

In order to be able to use *ADD* with probabilistic effects, one would need to split into as many plan graphs as there are leaves in a probabilistic action. To avoid this, we split each action in the domain theory into as many deterministic actions as the number of nonempty effect lists each representing a possible way the original

action would work (Fig. 2). By using the split actions, we can compute a good estimate of the number of actions needed to complete a plan. While the plan graph uses split actions, the plans in the search queue always contain the full original action so that the planner can correctly assess the probability of success. Our current ranking function uses this assessment to prefer plans with higher probability of success, and if there is a tie, the plan with less number of steps is preferred.

An important distinction between deterministic partial-order planning and probabilistic partial-order planning is multiple support for plan literals. In the deterministic case, an open condition is permanently removed from the list of flaws once it is resolved. In the probabilistic case, it can be *reopened* so that the planner can search for additional steps that increase the probability of the literal. We address this problem by employing *selective reopening* (SR) where we select a random total ordering of the plan; look at the state distribution after the execution of each step; and reopen only those conditions that are not guaranteed to be achieved. While plan assessment is costly for probabilistic plans, this is a one time cost incurred only on quasi-complete plans and we have observed that the benefit of avoiding extra plans in the search space far exceeds the computational overhead incurred.

It is important to note that neither the split actions nor the selective reopening technique change the base soundness and completeness properties of the Buridan algorithm. The split actions are only used in the relaxed plan graph, and the reopening technique does not block any alternatives from being sought as they would already be covered by a plan in the search queue.

## Conclusion and Future Work

We presented Probapop, a partial-order probabilistic planner. We described distance-based and probabilistic condition based heuristics for partial-order probabilistic planning. We informally noted that neither the split actions nor the selective reopening technique change the base soundness and completeness properties of the Buridan algorithm.

Probapop is different than policy generating planners such as Spudd(Hoey *et al.* 1999) and Gpt(Bonet & Geffner 2000) in the sense that it generates plans. Given a planning problem, Probapop returns a sequence of steps that achieve the goal with a probability that meets or exceeds the specified threshold. The plan generated does not rely on sensing actions in order to be executed. Our future work involves adding the capability to deal with partially observable domains to Probapop.

## References

- Bertoli, P.; Cimatti, A.; and Roveri, M. 2001. Heuristic search + symbolic model checking = efficient conformant planning. In *Proc. 18th Intl. Joint Conf. on Artificial Intelligence*, 467–472.
- Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *AIJ* 90:281–300.
- Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *Proc. 5th European Conf. on Planning (ECP'99)*.
- Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Proc. 5th Intl. Conf. AI Planning & Scheduling*, 52–61.
- Ferraris, P., and Giunchiglia, E. 2000. Planning as satisfiability in nondeterministic domains. In *Proc. 17th Nat. Conf. Artificial Intelligence*, 748–754.
- Hansen, E. A., and Feng, Z. 2000. Dynamic programming for POMDPs using a factored state representation. In *Proc. 5th Intl. Conf. AI Planning & Scheduling*, 130–139.
- Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUDD: Stochastic planning using decision diagrams. In *Proc. 15th Conf. Uncertainty in AI*.
- Hyafil, N., and Bacchus, F. 2003. Conformant probabilistic planning via csps. In *Proc. 13th Intl. Conf. Automated Planning & Scheduling*.
- Karlsson, L. 2001. Conditional progressive planning under uncertainty. In *Proc. 18th Intl. Joint Conf. on Artificial Intelligence*, 431–436.
- Kushmerick, N.; Hanks, S.; and Weld, D. S. 1995. An algorithm for probabilistic planning. *AIJ* 76:239–286.
- Majercik, S. M., and Littman, M. L. 1999. Contingent planning under uncertainty via stochastic satisfiability. In *Proc. 16th Nat. Conf. Artificial Intelligence*, 549–556.
- McDermott, D. 1999. Using regression-match graphs to control search in planning. *AIJ* 109(1-2):111–159.
- Nguyen, X., and Kambhampati, S. 2001. Reviving partial order planning. In *Proc. 18th Intl. Joint Conf. on Artificial Intelligence*, 459–464.
- Onder, N., and Pollack, M. E. 1999. Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In *Proc. 16th Nat. Conf. Artificial Intelligence*, 577–584.
- Penberthy, J. S., and Weld, D. S. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. Third Intl. Conf. Principles of Knowledge Representation & Reasoning*, 103–114.
- Smith, D. E.; Frank, J.; and Jonsson, A. K. 2000. Bridging the gap between planning and scheduling. *Knowledge Engineering Review* 15(1).
- Younes, H. L., and Simmons, R. G. 2002. On the role of ground actions in refinement planning. In *Proc. 6th Intl. Conf. AI Planning & Scheduling*, 54–61.

# Probabilistic Reachability Analysis for Structured Markov Decision Processes

Florent Teichteil-Königsbuch and Patrick Fabiani

ONERA-DCSD 2 Avenue Édouard-Belin  
31055 Toulouse, France  
(florent.teichteil,patrick.fabiani)@cert.fr

## Abstract

We present a stochastic planner based on Markov Decision Processes (MDPs) that participates to the probabilistic planning track of the 2004 International Planning Competition. The planner transforms the PDDL problems into factored MDPs that are then solved with a structured policy iteration algorithm. A probabilistic reachability analysis is performed, approximating the MDP solution over the reachable states subspace, in order to restrict the search space and allow a subsequent heuristic search.

## Introduction

We present a planner based on Markov Decision Processes (MDPs) (Puterman 1994) to participate in the probabilistic planning track of the International Planning Competition at ICAPS'04. MDPs provide a decision-theoretic framework for planning with uncertain actions effects. A MDP (Puterman 1994) is a Markov chain controlled by an agent. A control strategy associates to each state the choice of an action, whose result is a stochastic state. The Markov property means that the probability of arriving in a particular state after an action only depends on the previous state of the chain and not on the entire states history. Formally it is a tuple  $\langle S, A, T, R \rangle$  where  $S$  is the set of states,  $A$  is the set of actions,  $T$  and  $R$  are functions giving respectively the transition probabilities between states (depending on the chosen action) and the immediate or terminal rewards (depending on the starting state, the chosen action and the ending state). The most frequent optimisation criterion consists in maximizing the infinite horizon sum  $E(\sum_{t=0}^{\infty} \beta r_t)$  of expected rewards  $r_t$  discounted by a factor  $0 < \beta < 1$  that insures the convergence of algorithms, but can also be interpreted as a uncontrolled stopping probability between two time points.

The resolution of MDPs is based on dynamic programming and includes two classes of algorithms : value iteration and policy iteration. The first is an iteration on the value function associated with each state, that is to say the expected accumulated reward starting from this state. When the iterated value function stabilizes, the optimal value function is reached and the optimal policy follows. In the policy

iteration scheme, the current policy is assessed on the infinite horizon and improved locally at each iteration. The value of a policy  $\pi$  is solution of Bellman's equations (Bellman 1957) :

$$V^{\pi}(s) = \sum_{s' \in S} T(s, \pi(s), s') \cdot (R(s, \pi(s), s') + \beta V^{\pi}(s'))$$

Compared to value iteration, the policy iteration algorithm converges in fewer iterations, but each policy assessment stage may be computationally costly. A large discussion about criteria and resolution algorithms is proposed in (Puterman 1994).

## Motivations and issues

Nevertheless, classical exact algorithms (based on stochastic dynamic programming on an explicitly enumerated state space) are not effective enough for realistic applications that often have very large state spaces (Boutilier & Hanks 1999; Verfaillie, Garcia, & Péret 2003). Proposed techniques to solve such problems include approximating or learning methods (Bertsekas & Tsitsiklis 1995) where the computing cost and the error are both controlled. Other approaches exploit the natural structure of planning problems either by using compact factored representations (Boutilier & Hanks 1999; Boutilier, Dearden, & Goldszmidt 2000; Hoey *et al.* 2000), or by decomposing the state space in sub-regions (Hauskrecht *et al.* 1998; Dean & Lin 1995; Parr 1998) that enables a hierarchical resolution that is sometimes more effective.

Our initial motivations are to combine factored and enumerated state representations in probabilistic planning (Teichteil-Königsbuch & Fabiani 2004). The obtained hybrid MDP model exploits the problem structure in terms of both decomposition and factorization. This approach is adapted for stochastic planning problems involving both intermediate tasks planning and navigation planning. Tools are needed in order to restrict the search space to its useful part and allow an efficient heuristic search in useful regions.

## State space factorization

Our planner uses a compact factored representation of MDPs based on Algebraic Decision Diagrams (ADDs) (R.I. Bahar *et al.* 1993) and is inspired from (Hoey *et al.* 2000).

Since the problems of the stochastic planning track of the competition are given in the PPDDL 1.0 language (Younes & Littman 2003), we must translate the PPDDL problem definitions into ADDs-based MDP representation.

The factorization of the state space consist in a cross product involving state variables :  $S = \otimes_{i=1}^n x_i$ . It is a compact representation because the states are no longer enumerated in a list, but rather structured by the set of random state variables:  $x_{i=1}^n$ . Such variables enable to process sets of states, instead of individual states, whenever useful. For each action, the transition probability into a given state is no longer given as a function of the individual initial state but now depends conditionnally on the state variables. Therefore, they can be represented either as Dynamic Bayesian Networks (Dean & Kanazawa 1989) or with probabilistic STRIPS operators (Dearden & Boutilier 1997).

### Dynamic Bayesian Networks (DBNs)

A factored MDP can be represented by use of a set of action networks. For each action, an action network (which is a DBN) represents the probabilistic effects and rewards obtained on the variables after the action has been performed (*post-action variables*), conditionally to the possible values of the variables before the action is applied (*pre-action variables*). There can exist *diachronic arcs*, directed from pre-action variables to post-action variables, and *synchronic arcs* encoding for dependences (correlations) between post-action variables. Such DBNs represent the factored conditional (controlled) transition probabilities within the state space, encoded as conditional probabilities of obtaining the post-action variables knowing the pre-action variables. The corresponding immediate rewards are directly associated to the possible transitions. These data are stored respectively in a Conditional Probability Table and in a Conditional Reward Table. Such data structures can be represented either as a set of decision trees (Boutilier, Dearden, & Goldszmidt 2000) or as a set of Algebraic Decision Diagrams (ADDs) (Hoey *et al.* 2000). Although ADDs only deal with binary variables (boolean values), they are in most cases much more effective than decision trees. Non-binary variables are then encoded using a number of boolean variables (Hoey *et al.* 2000).

### Resolution scheme

The resolution scheme corresponding to factored MDPs, named *Decision-Theoretic Regression*, avoid the explicit enumeration of all states at each iteration. The corresponding algorithms are structured versions of the classical MDPs resolution algorithms, which use algebraic operations defined on decision trees, or ADDs, in order to solve Bellman's equations for these data structures. For instance, using ADDs, the conditional probabilities ADDs of the possible actions (Probability ADDs) and conditional reward values ADDs of the possible actions (Reward ADDs) are combined in order to provide both Value Function ADDs and Policy ADDs on the factored state space. The algorithms directly perform the operations on ADDs (the same on decision trees naturally). The SPI algorithm (Boutilier, Dearden, & Goldszmidt 2000) is a value iteration scheme based

on decision trees. The SPUDD and APRICODD algorithms (Hoey *et al.* 2000), based on ADDs, are respectively value iteration and approximated value iteration algorithms for factored MDPs. As SPUDD, we use the CUDD package (Somenzi 1998) as an ADD library in our planner.

### Policy iteration with ADDs

However, our planner rather implements a structured version of the modified policy iteration. As a matter of fact, we did not find any implementation of the policy iteration scheme based on the CUDD package. To our experience, the CUDD package does not provide directly a number of operations that appear as useful for policy iteration. For instance, policy evaluation requires an operation on the current Policy ADD  $\Pi$ , which replaces each leaf labelled by the number of an action  $a$  (Policy ADDs have leaves labelled by action numbers) with the Reward ADD  $R_a$  of this action  $a$ , and replaces the other leaves by 0. Let us call *ConcatActionRewardADDPolicy*( $\Pi, a$ ) such an operation that outputs an ADD  $R_a^\Pi$  having the same leaves values as  $R_a$  when applicable according to  $\Pi$ , 0 otherwise.  $R_\Pi =_{a \in A} R_a^\Pi$  is the immediate reward ADD applying  $\Pi$  over the state space.

$R_\pi \leftarrow 0$

**For**  $a$  **from** 1 **to**  $|A|$  **do**

$R_a^\Pi \leftarrow \text{ConcatActionRewardADDPolicy}(\Pi, a)$   
 $R_\pi \leftarrow R_\Pi + R_a^\Pi$

Similarly, we need a *ConcatActionProbADDPolicy*( $\Pi, a$ ) to compute the probability ADDs  $P_a^\Pi$  that applies the Probability ADD  $P_a$  of action  $a$  whenever applicable according to  $\Pi$ , and 0 otherwise.  $P_\Pi =_{a \in A} P_a^\Pi$  is the transition Probability ADD over the state space  $S$  applying  $\Pi$ . The implemented version of these operations could possibly be improved by writing new low-level procedures for the CUDD package.

### Correlations

The resolution of factored MDPs can sometimes be specifically improved, depending on the specific features of the problem. For instance, dealing with correlations between post-action variables in action networks (*synchronic arcs*) may be an issue. In (Boutilier, Dearden, & Goldszmidt 2000), it is proposed to replace such parasitic post-action variables in decision trees (or ADDs) by modified subtrees containing only pre-action variables. However, this complex operation can be avoided. This is done in our planner by using a single *complete action diagram* per action network (Hoey *et al.* 2000) that represents the product of the conditional probabilities of obtaining the post-action variables knowing the pre-action variables; as a matter of fact, the correlations in that case are implicit and they do not require a specific treatment.

### Probabilistic Reachability Analysis and Heuristic Search

Coping with large state spaces is a really challenging issue when dealing with realistic problems. This problem has been addressed from at least two different points of view in the literature :

- Reachability analysis : when the initial state is known, a reachability analysis allows to dismiss state variables combinations (sets of states) corresponding to states that will never be reached or traversed. For example, the algorithm REACHABLEK proposed in (Boutilier, Brafman, & Geib 1998) enables to push away from trees (or ADDs in the same way) the nodes corresponding to states that are not reachable when starting from a given starting state.
- heuristic search : an heuristic search algorithm can be used in order to speed up the optimization algorithms, either by producing good initialization values for iterative optimization, or by leading the optimization algorithm to run on more useful regions of the state space. For example, the algorithm proposed in (Feng & Hansen 2001) does both and guarantees to converge towards the optimal solution by using an admissible heuristic. It performs value iteration on a restriction  $E$  of the state space. It uses a lower bound estimation as a heuristic initial value assigned on the “fringe” states on the border of  $E$  for value iteration on the states of  $E$ . This heuristic also determines the “expansion” of  $E$  via a reachability analysis using the current “partial” policy  $\Pi$  given by policy iteration at this stage.

The meeting point of both points of view is reached when the heuristic search is based on a reachability analysis. In our planner we perform a probabilistic reachability analysis on the problem. We use it in the policy iteration scheme in order to provide an initial partial policy. We also use it to restrict the resolution algorithm on a useful subspace of the state space. These aspects of the resolution scheme are still under development and require further work.

## Conclusion

We have presented our probabilistic planner which is based on Factored Markov Decision Processes (MDPs) as a decision-theoretic framework for planning under uncertainty. The work described in this short paper is still incomplete at this time, but will be completed for the probabilistic planning track of the International Planning Competition at ICAPS'04. We expect the competition to lead to improvements of our algorithms, to be used later in a more general framework combining factored and enumerated state representations. Such an hybrid MDP model allows to take advantage of the problem structure in terms of both (geographical) decomposition and factorization. It is more dedicated to stochastic planning problems involving both intermediate tasks planning and navigation planning, such as exploration missions. This research is part of the autonomous helicopter project *ReSSAC* project at ONERA (<http://www.cert.fr/dcsd/RESSAC>).

## References

- Bellman, R. 1957. *Dynamic Programming*. Princeton, NJ: Princeton University Press.
- Bertsekas, D. P., and Tsitsiklis, J. N. 1995. Neuro-dynamic programming: an overview. In *Proceedings of the 34th Conference on Decision and Control*, 560–564.
- Boutilier, C., and Hanks, T. D. S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *J. of Artificial Intelligence Research* 11:1–94.
- Boutilier, C.; Brafman, R. I.; and Geib, C. 1998. Structured reachability analysis for Markov decision processes. In *Uncertainty in Artificial Intelligence*, 24–32.
- Boutilier, C.; Dearden, R.; and Goldszmidt, M. 2000. Stochastic dynamic programming with factored representations. *Artificial Intelligence* 121(1-2):49–107.
- Dean, T., and Kanazawa, K. 1989. A model for reasoning about persistence and causation. *Computational Intelligence* 5(3):142–150.
- Dean, T., and Lin, S.-H. 1995. Decomposition techniques for planning in stochastic domains. In *Proceedings of the 14th IJCAI 1995*, 1121–1129.
- Dearden, R., and Boutilier, C. 1997. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence* 89:219–283.
- Feng, Z., and Hansen, E. 2001. Symbolic heuristic search for factored markov decision processes. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, 455–460. Edmonton, Canada: AAAI Press / The MIT Press.
- Hauskrecht, M.; Meuleau, N.; Kaelbling, L. P.; Dean, T. L.; and Boutilier, C. 1998. Hierarchical solution of markov decision processes using macro-actions. In *Proceedings of 14th Conf. UAI 1998*, 220–229.
- Hoey, J.; St.Aubin, R.; Hu, A.; and Boutilier, C. 2000. Optimal and approximate stochastic planning using decision diagrams. Technical Report TR-2000-05, University of British Columbia.
- Parr, R. 1998. Flexible decomposition algorithms for weakly coupled markov decision problems. In *Proceedings of 14th Conf. UAI 1998*, 422–430.
- Puterman, M. L. 1994. *Markov Decision Processes*. John Wiley & Sons, INC.
- R.I. Bahar; E.A. Frohm; C.M. Gaona; G.D. Hachtel; E. Macii; A. Pardo; and F. Somenzi. 1993. Algebraic Decision Diagrams and Their Applications. In *IEEE / ACM International Conference on CAD*, 188–191. Santa Clara, California: IEEE Computer Society Press.
- Somenzi, F. 1998. Cudd: Cu decision diagram package. Technical report, University of Colorado at Boulder.
- Teichteil-Knigsbuch, F., and Fabiani, P. 2004. Un modèle hybride en planification probabiliste d’exploration autonome. In *Proceedings RFIA'04*.
- Verfaillie, G.; Garcia, F.; and Péret, L. 2003. Deployment and Maintenance of a Constellation of Satellites: a Benchmark. In *Proceedings of ICAPS'03 Workshop on Planning under Uncertainty and Incomplete Information*.
- Younes, H. L., and Littman, M. L. 2003. Ppddl 1.0: An extension to pddl for expressing planning domains with probabilistic effects.

# Learning Reactive Policies for Probabilistic Planning Domains

SungWook Yoon and Alan Fern and Robert Givan

Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907 USA  
{sy, afern, givan}@purdue.edu

## Abstract

We present a planning system for selecting policies in probabilistic planning domains. Our system is based on a variant of approximate policy iteration that combines inductive machine learning and simulation to perform policy improvement. Given a planning domain, the system iteratively improves the best policy found so far until no more improvement is observed or a time limit is exceeded. Though this process can be computationally intensive, the result is a reactive policy, which can then be used to quickly solve future problem instances from the planning domain. In this way, the resulting policy can be viewed as a domain-specific reactive planner for the planning domain, though it is discovered with a domain-independent technique. Thus, the initial cost of finding the policy is amortized over future problem-solving experience in the domain. Due to the system's inductive nature, there are no performance guarantees for the selected policies. However, empirically our system has shown state-of-the-art performance in a number of benchmark planning domains, both deterministic and stochastic.

## Introduction

We view a planning domain (e.g. as specified via PPDDL) as a Markov Decision Process (MDP) where there is an MDP state for each possible problem instance in the domain. Viewed as such, a solution to the MDP, i.e. a policy, is a mapping from problem instances to domain actions. For goal-based domains, such a policy can be viewed as specifying what action to take given the current domain state and current goal. A good policy will select actions so as to minimize the expected cost of reaching the goal.

Typically the MDP corresponding to a PPDDL domain has far too many states to support solution via flat state-space MDP techniques. To deal with large state spaces we base our system on a form of approximate policy iteration (API), which does not rely on state-space enumeration. Most existing frameworks for API (e.g. (Bertsekas & Tsitsiklis 1996)) represent policies indirectly via value functions and use machine learning to select value function approximations. However, in many domains, particularly those with relational (first-order) structure, representing and learning

value functions is much more complicated than representing and learning policies directly. Based on this observation, our system utilizes a new variant of API (Fern, Yoon, & Givan 2003), which represents policies directly as state/action mappings.

The performance of our system depends on two critical issues. First, we must provide a policy language and associated learner that allow the system to find approximations of good policies. Second, for complex domains, it is necessary to provide a mechanism to bootstrap the API process. Below we describe the choices we have made to deal with these issues in our current system.

In what follows we first provide an overview of API. Next we discuss the policy representation language and learning technique used in our system. Finally, we give an overview of our bootstrapping technique. A more detailed treatment of our algorithms can be found in (Fern, Yoon, & Givan 2003; 2004).

## Approximate Policy Iteration

Figure 1 shows the core components of our system's API engine. Each iteration of API consists of two primary stages: *policy evaluation* and *policy selection*. Intuitively, policy evaluation uses simulation to produce a training set that describes an improved policy with respect to the current policy. Policy selection then uses machine learning to find an approximation of the improved policy based on the training set. Thus, if we are given a current policy and then apply these steps in sequence, the result is an (approximately) improved policy. Our system iterates these steps until no more improvement is observed.

**Policy Evaluation.** Policy evaluation is carried out via the simulation technique of *policy rollout* (Bertsekas & Tsitsiklis 1996). The policy-rollout component first draws a set of problem instances (which can also be viewed as MDP states) from the provided problem generator.<sup>1</sup> Next, for each problem instance  $I$  and each action  $a$  available in  $I$ , simulation is used to estimate the Q-value  $Q(I, a, \pi)$  of the current

<sup>1</sup>Even when a problem generator is not provided for a planning domain, we can still use API to solve individual problem instances. Given an individual problem instance to be solved, we simply create a trivial problem generator that always returns that problem instance.

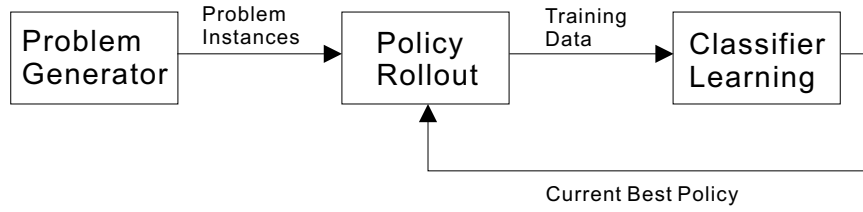


Figure 1: Block diagram of approximation policy iteration. We assume each planning domain provides a problem generator and our goal is to produce a policy that performs well on problem instances drawn from the generator. Given the current best policy, the policy-rollout component creates a training set that describes an improved policy as evaluated on problems drawn from the generator. The classifier learner then analyzes this training set and outputs an approximately improved policy.

policy  $\pi$ , which is simply the expected value of taking action  $a$  in  $I$  and then following  $\pi$  until a terminating state is reached or a horizon limit is exceeded.

It is straightforward to compute a new improved policy  $\pi'$  from the Q-values of policy  $\pi$ . In particular, it is a basic property of MDPs that  $\pi'$  defined as  $\pi'(I) = \operatorname{argmax}_a Q(I, a, \pi)$  is guaranteed to improve upon  $\pi$ , if improvement is possible. Thus, for each of our sample problem instances, the estimates of  $Q(I, a, \pi)$  can be used to calculate  $\pi'(I)$ , that is an “improved action” for problem instance  $I$ . Intuitively, the pairs  $\langle I, \pi'(I) \rangle$  can be viewed as training examples for learning an approximation of  $\pi'$ . To support such learning, the output of the policy-rollout component is a training set, with one training instance  $\langle I, Q(I, a_1, \pi), \dots, Q(I, a_n, \pi) \rangle$  for each instance  $I$  drawn from the problem generator. Please refer to (Fern, Yoon, & Givan 2003) for more details.

**Policy Selection.** Policy selection is carried out by the classifier-learning component of our system. Note that a policy can be viewed as a classifier that maps problem instances (i.e. states) to actions. The training set obtained from policy rollout is used to learn an (approximately) improved policy. Given a language for compactly representing policies, the job of the classifier learner is to select a policy within that language that chooses actions with high Q-value for problem instances in the training set. With a proper language bias, such policies also tend to select good actions in problem instances outside of the training set. In the next section we give an overview of the policy description language and the corresponding learner used in our system.

**Compute Time.** In our current system, the computation time of API is mostly consumed by generating training sets via policy rollout. This is particularly the case for domains where problem instances contain many ground actions, as multiple trajectories must be simulated for each ground action in each problem instance encountered. Presently the rollout component is implemented in Scheme, hence one way to significantly improve runtime is to provide a C implementation. We are also working to exploit the independence of the rollout trajectories with a parallel implementation. If completed, this speedup may be in effect for our competition entry.

## Representing and Learning Policies

For API to succeed, we must provide an adequate language for representing good policies in a domain, and an associated learner that can find good policies, in that language, based on the guidance provided by the rollout training sets.

One of our primary interests is in applying our system to relationally structured planning domains, such as the blocks world, where problem instances are described by specifying a domain of objects (e.g. a set of blocks) and relations among the objects. Thus, it is critical that we provide a policy language that leverages the relational structure in order to generalize across problem instances with different sets of objects. For example, our language needs to represent policies that can be applied to any problem instance of the blocks worlds, regardless of the number and identity of blocks. In order to represent such “generalized policies” we draw upon ideas from the knowledge-representation community, using a language based on taxonomic syntax.

**Policy Representation.** Our policy representation is an ordered list of rules. The head of each rule is a variablized action type such as **pickup**(?a). The body of each rule specifies a conjunction of constraints on the “object variables” in the head, which indicate when an action should be applied. Given a problem instance, we say that a rule suggests an action if: 1) the action is the result of replacing the object variables in the head with objects from the problem instance, and 2) those objects satisfy the appropriate constraints in the body. The action selected by an ordered list of rules (i.e. a policy) is equal to the action chosen by the earliest rule that selects an action.

The object constraints in a rule’s body are represented via taxonomic syntax expressions, which are constructed from the predicate symbols of the planning domain and object variables in the rule’s head. As an example policy, consider a blocks-world domain where the goal is always to clear off block  $A$ . We can represent an optimal policy in our taxonomic representation as follows.

**pickup**(?a) : (?a  $\in$  on\* A)  $\wedge$  (?a  $\in$  clear)  
**putdown**(?a) : ?a  $\in$  holding

The first rule indicates that we should “pick up a clear block which is above block A”. The second rule says that we should “put down any block that is being held”.

For a detailed description of the syntax and semantics of

our policy language please refer to the appendix of (Fern, Yoon, & Givan 2004).

**Learning.** Recall that each training instance is of the form  $\langle I, Q(I, a_1, \pi), \dots, Q(I, a_n, \pi) \rangle$ , where  $I$  is a problem instance and the  $Q(I, a_i, \pi)$  are the associated Q-values. The goal of the learner is to select a list of rules such that the actions chosen by the corresponding policy results in high Q-value over the training data. Ideally the learned policy should always select an action corresponding to the largest Q-value.

We use a simple greedy covering strategy for learning lists of taxonomic rules. We add one rule to the list at a time until the resulting policy covers all of the training data (i.e. the policy selects an action for every problem instance in the training data). Each rule is learned by greedily adding object constraints to the body according to a heuristic measure that attempts to balance the coverage and quality of a rule. For more information on the learner, please refer to (Yoon, Fern, & Givan 2002) and (Fern, Yoon, & Givan 2003).

### Bootstrapping from Random Walks

API must be initialized with a base policy from which iterative policy improvement begins. Since our objective is to have a domain-independent system, we use the random policy as the default base policy in our system. However, for many planning domains it is unlikely that a random policy will achieve any non-trivial reward in problem instances drawn from the provided problem generator. For example, in a blocks world with even a relatively small number of blocks, it is unlikely that a random policy will achieve the goal configuration. As a result, in such domains, API will tend to fail when initialized with a random base policy. The primary reason for the failure is that the Q-values for each action under the random policy will tend to be equal. Thus, the rollout training set, which is based on the Q-values, will not provide the learner with useful guidance as to what actions are desirable.

Our current approach to this problem is to utilize a new bootstrapping technique (Fern, Yoon, & Givan 2004). Rather than initially drive API with the original problem generator (which generates difficult problems), we instead automatically construct a new problem generator that generates easier problems. We then increase the problem difficulty in accordance with the quality of the current best policy found by API. Below we describe this process for goal-based domains. Our current system does not provide a bootstrapping mechanism for non-goal-based domains.

We generate problem instances of varying difficulty by performing random walks in the planning domain. To construct a single problem instance from a planning domain, we first draw a problem instance from the original problem generator. In a goal-based setting, such a problem instances will specify an (initial) domain state  $s$  and a goal. Next, starting at  $s$ , we take a sequence of  $n$  random actions (i.e. an  $n$ -step random walk) and observe the resulting state  $g$ . We construct a new problem instance with initial state  $s$  and goal  $g$ . When  $n$  is small, such problem instances are relatively easy to solve and we can learn a policy to solve all such problem instances using API starting with a random base policy.

Once we learn a policy for “random-walk problems” with small  $n$ , we increase the value of  $n$  until the current policy performs poorly and then continue to apply API using the more difficult problem distribution. This process of iteratively increasing  $n$  and then applying API continues until we either achieve a policy that performs well on the original problem distribution or no more improvement is observed. For more details and empirical results please see (Fern, Yoon, & Givan 2004).

### Acknowledgments

This work was supported in part by NSF grants 9977981-IIS and 0093100-IIS.

### References

- Bertsekas, D. P., and Tsitsiklis, J. N. 1996. *Neuro-Dynamic Programming*. Athena Scientific.
- Fern, A.; Yoon, S.; and Givan, R. 2003. Approximate policy iteration with a policy language bias. In *NIPS*.
- Fern, A.; Yoon, S.; and Givan, R. 2004. Learning domain-specific control knowledge from random walks. In *ICAPS*.
- Yoon, S.; Fern, A.; and Givan, R. 2002. Inductive policy selection for first-order MDPs. In *UAI*.



## List of Authors

<i>Bonet, Blai</i> .....	74	<i>Schaeffer, Jonathan</i> .....	15
<i>Botea, Adi</i> .....	15	<i>Serina, Ivan</i> .....	33
<i>Camilleri, Guy</i> .....	21	<i>Silva, Fabiano</i> .....	27
<i>Casilho, Marcos</i> .....	27	<i>Skvortsova, Olga</i> .....	83
<i>Chen, Yixin</i> .....	30	<i>Smith, Amanda</i> .....	24
<i>Coles, Andrew</i> .....	24	<i>Tang, Minh</i> .....	53
<i>Edelkamp, Stefan</i> .....	2,7	<i>Teichteil-Königsbuch, Florent</i> .....	89
<i>Englert, Roman</i> .....	7	<i>Thiébaux, Sylvie</i> .....	7, 80
<i>Enzenberger, Markus</i> .....	15	<i>Toninelli, Paolo</i> .....	33
<i>Fabini, Patrick</i> .....	89	<i>Trüg, Sebastian</i> .....	7
<i>Fern, Alan</i> .....	92	<i>U, Senthil</i> .....	46
<i>Geffner, Héctor</i> .....	59, 74	<i>Vidal, Vincent</i> .....	56, 59
<i>Gerevini, Alfonso</i> .....	33	<i>Wah, Benjamin W.</i> .....	30
<i>Givan, Robert</i> .....	64, 92	<i>Whelan, Garrett. C</i> .....	86
<i>Gretton, Charles</i> .....	80	<i>Yoon, SungWook</i> .....	92
<i>Guedes, André</i> .....	27	<i>Younes, Håkan L. S.</i> .....	68, 70
<i>Halsey, Keith</i> .....	35	<i>Zalakiet, Joseph</i> .....	21
<i>Hansen, Eric A.</i> .....	61, 77	<i>Zhengzhu Feng</i> .....	77
<i>Haslum, Patrik</i> .....	38	<i>Zhou, Rong</i> .....	61
<i>Helmert, Malte</i> .....	41	<i>Zhu, Lin</i> .....	64
<i>Hoffmann, Jörg</i> .....	2,7	<i>van den Briel, Menkes</i> .....	18
<i>Hsu, Chih-Wei</i> .....	30		
<i>Kambhampati, Subbarao</i> .....	18		
<i>Karabaev, Eldar</i> .....	83		
<i>Kautz, Henry</i> .....	44		
<i>Kavuluri, Bharat Ranjan</i> .....	46		
<i>Künzle, Luis</i> .....	27		
<i>Li, Li</i> .....	86		
<i>Lima, Tiago</i> .....	27		
<i>Liporace, Frederico</i> .....	7		
<i>Littman, Michael</i> .....	68, 70		
<i>Müller, Martin</i> .....	15		
<i>Mali, Amol D.</i> .....	53		
<i>Marynowski, João</i> .....	27		
<i>McDermott, Drew</i> .....	48		
<i>Montaño, Razer</i> .....	27		
<i>Onder, Nilufer</i> .....	86		
<i>Parker, Eric</i> .....	51		
<i>Price, David</i> .....	80		
<i>Richter, Silvia</i> .....	41		
<i>Saetti, Alessandro</i> .....	33		
<i>Sanchez, Javier</i> .....	53		