ICAPS 2006

The English Lake District, Cumbria, UK

Fifth International Planning Competition

Alfonso Gerevini
Università di Brescia, Italy

Blai Bonet
Universidad Simón Bolívar, Venezuela

Bob Givan
Purdue University, USA

IPC

ICAPS 2006

The English Lake District, Cumbria, UK

Fifth International Planning Competition

Alfonso Gerevini
Università di Brescia, Italy

Blai Bonet
Universidad Simón Bolívar, Venezuela

Bob Givan
Purdue University, USA

IPC

ICAPS 2006
International Planning Competition

Table of contents

Preface

The international planning competition is a biennial event with several goals, including analyzing and advancing the state-of-the-art in automated planning systems; providing new data sets to be used by the research community as benchmarks for evaluating different approaches to automated planning; emphasizing new research issues in planning; promoting the acceptance and applicability of planning technology.

The fifth international planning competition, IPC-5 for short, has attracted many researchers. As in the fourth competition, IPC-5 and its organization is split into two parts: the Deterministic Track, that considers fully deterministic and observable planning (previously also called "classical" planning), and the Probabilistic Track, that considers non deterministic planning.

The deterministic part is organized by two groups of people: an organizing committee, that is in charge of the various activities for running the competition, and a consulting committee, that was mainly involved in the early phase of the organization to discuss an extension to the language of the competition (PDDL) to be used in IPC-5.

The deterministic part of IPC-5 has two main novelties with respect to previous competition. Firstly, while considering the CPU-time, we intend to give more emphasis to the importance of plan quality, as defined by the problem plan metric. Partly motivated by this reason, we significantly extended PDDL to include some new constructs, aiming at a better characterization of plan quality by allowing the user to express strong and "soft" constraints about the structure of the desired plans, as well as strong and soft problem goals. The new language, called PDDL3, was developed in strict collaboration with Derek Long, a member of the IPC-5 consulting committee.

In PDDL3.0, the version of PDDL3 used in the competition, we can express problems for which only a subset of the goals and plan trajectory constraints can be achieved (because they conflict with each other, or because achieving all them is computationally too expensive), and where the ability to distinguish the importance of different goals and constraints is critical. A planner should try to find a solution that satisfies as many soft goals and constraints as possible, taking into account their importance and their computational costs. Soft goals and constraints, or preferences, as they are called in PDDL3.0, are taken into account by the plan metric, which can give a penalty for failure to satisfy each of the preferences (or, conversely, a bonus for satisfying them). The extensions made in PDDL3.0 seem to have gained fairly wide acceptance, with more than half the competing planners in the deterministic track supporting at least some of the new features.

Another novelty of the deterministic part of IPC-5 which required considerable efforts concerns the test domains: we designed five new planning domains, together with a large collection of benchmark problems. In order to make PDDL3.0 language more accessible to the competitors, for each test domain, we developed various variants using different fragments of PDDL3.0 with increasing expressiveness. In addition, we re-used two domains from previous competitions, extended with new variants including some of the features of PDDL3.0. The IPC-5 test domains have different motivations. Some of them are inspired by real world applications; others are aimed at exploring the applicability and effectiveness of automated planning for new applications or for problems that have been investigated in other field of computer science; while the domains from previous competitions are used as sample references for measuring the advancement of the current planning systems with respect to the existing benchmarks.

The probabilistic track of the competition appeared for the first time in the fourth edition of the competition in 2004. The probabilistic track consists of probabilistic planning problems with complete observability specified in the PPDDL language. The focus of the competition is in planners that can deliver real-time decision making as opposed to complete policies. The planners are evaluated using the client/server architecture developed for the probabilistic track of IPC-4. Thus, any type of planner can enter the competition as long as it is able to choose and send actions to the server. The planners are evaluated in a number of episodes for each instance problem from which an estimate of the average cost to the goal of planner's policy is computed. The planners are then ranked using such scores.

This year's competition includes, for the first time, a conformant planning subtrack within the probabilistic track. In conformant planning, the planners are faced with non-deterministic planning problems and required to output a contingency-safe and linear plan that solves the problem. Planners in this subtrack are evaluated in terms of the CPU time required to output a valid plan.

We have included novel and interesting domains in the probabilistic and conformant tracks which aims to reveal interesting tradeoffs in non-deterministic planning. The domain codifications are as simple as possible trying to avoid complex syntactic constructs such as nested conditional effects, disjunctive preconditions and goals, etc. Indeed, some domains are grounded codifications (as some domains in the deterministic track of IPC-4), while others are 'lifted' first-order codifications of problems, which can be exploited by some of the planners. We have included problem generators for almost all the domains so to allow the competitors to tune their planners. The competition benchmark consisted of a set of domains for practice and another set for the actual competition.

In the deterministic track of IPC-5, there are 14 competing teams (initially they were 18, but 4 of them had to withdraw their planners during the competition), each of which can participate with at most two planners (or variants of the same planner), and 40 participating researchers from various universities and research institutes in Europe, USA, Canada and India.

The probabilistic track consists of 8 teams divided into 2 groups of 4 teams each for probabilistic and conformant planning respectively. The teams are from various universities and research institutes in USA, Canada, Europe and Australia.

At the time of writing the competition is still running. The results will be announced at ICAPS'06 and made available from the deterministic and probabilistic websites of the competition. This booklet contains the abstracts of the IPC-5 planners that are currently running the competition tests. The descriptions of the planners may be in many cases preliminary, since the systems continue to evolve as they are faced with new problem domains.

The planner abstracts of the deterministic part of IPC-5 are preceded by an extended abstract describing the main features of PDDL3.0, which was distributed about six month before starting the competition, and by an extended abstract giving a short description of the benchmark domains.

The organizing committees of both tracks would like to send their best wishes and a great thanks to all the competing teams - it is mainly their hard efforts that make the competition such an exciting event!

Blai Bonet (Co-Chair Probabilistic Track)
Alfonso Gerevini (Chair Deterministic Track)
Bob Givan (Co-Chair Probabilistic Track)

Organizers (Deterministic track)

- *Yannis Dimopoulos - University of Cyprus (Cyprus)*
- *Alfonso Gerevini (chair) - University of Brescia (Italy)*
- *Patrik Haslum - Linköping University (Sweden)*
- *Alessandro Saetti - University of Brescia (Italy)*

Organizers (Probabilistic track)

- *Blai Bonet (co-chair) - Universidad Simn Bolvar (Venezuela)*
- *Robert Givan (co-chair) - Purdue University (U.S.A.)*

Consulting Committee (Deterministic Track)

- *Stefan Edelkamp*
- *Maria Fox*
- *Joerg Hoffmann*
- *Derek Long*
- *Drew McDermott*
- *Len Schubert*
- *Ivan Serina*
- *David Smith*
- *Dan Weld*

Consulting Committee (Probabilistic Track)

- *Hector Geffner*
- *Sylvie Thiebaux*

Plan Constraints and Preferences in PDDL3

The Language of the Deterministic Part of the Fifth International Planning Competition

Extended Abstract

Alfonso Gerevini$^+$ **and Derek Long***

$^+$ Department of Electronics for Automation, University of Brescia (Italy), gerevini@ing.unibs.it
* Department of Computer and Information Sciences, University of Strathclyde (UK), derek.long@cis.strath.ac.uk

Abstract

We propose an extension to the PDDL language, called PDDL3.0, that aims at a better characterization of plan quality by allowing the user to express strong and soft constraints about the structure of the desired plans, as well as strong and soft problem goals. PDDL3.0 was the reference language of the 5th International Planning competition (IPC-5). This paper contains most of the document about PDDL3.0 that was discussed by the Consulting Committee of IPC-5, and then distributed to the IPC-5 competitors.

Introduction

The notion of plan quality in automated planning is a practically very important issue. In many real-world planning domains, we have to address problems with a large set of solutions, or with a set of goals that cannot all be achieved. In these problems, it is important to generate plans of *good or optimal quality* achieving all problem goals (if possible) or some subset of them.

In the previous International planning competitions, the plan generation CPU-time played a central role in the evaluation of the competing planners. In the fifth International planning competition (IPC-5), while considering the CPU-time, we would like to give greater emphasis to the importance of plan quality. The versions of PDDL used in the previous two competitions (PDDL2.1 and PDDL2.2) allow us to express some criteria for plan quality, such as the number of plan actions or parallel steps, and relatively complex plan metrics involving plan makespan and numerical quantities. These are powerful and expressive in domains that include metric fluents, but plan quality can still only be measured by plan size in the case of propositional planning. We believe that these criteria are insufficient, and we propose to extend PDDL with new constructs increasing its expressive power about the plan quality specification.

The proposed extended language allows us to express *strong and soft constraints on plan trajectories* (i.e. constraints over possible actions in the plan and intermediate states reached by the plan), as well as *strong and soft problem goals* (i.e. goals that must be achieved in any valid plan, and goals that we desire to achieve, but that do not have to be necessarily achieved). Strong constraints and goals must be satisfied by any valid plan, while soft constraints and goals express desired constraints and goals, some of which may be more preferred than others. Informally, in planning with soft constraints and goals, the best quality plan should satisfy "as much as possible" the soft constraints and goals according to the specified preference relation distinguishing alternative feasible plans (satisfying all strong constraints and goals). While soft constraints have been extensively studied in the CSP literature, only very recently has the planning community started to investigate them (Brafman & Chernyavsky 2005; Briel *et al.* 2004; Delgrande, Schaub, & Tompits 2005; Miguel, Jarvis, & Shen 2001; Smith 2004; Son & Pontelli 2004), and we believe that they deserve more research efforts.

The following are some informal examples of plan trajectory constraints and soft goals. Additional formal examples will be given in the next section.

Examples in a blocksworld domain: *a fragile block can never have something above it, or it can have at most one block on it*; *we would like that the blocks forming the same tower always have the same colour*; *in some state of the plan, all blocks should be on the table*.

Examples in a transportation domain: *we would like that every airplane is used* (instead of using only a few airplanes, because it is better to distribute the workload among the available resources and limit heavy usage); *whenever a ship is ready at a port to load the containers it has to transport, all such containers should be ready at that port*; *we would like that at the end of the plan all trucks are clean and at their source location*; *we would like no truck to visit any destination more than once*.

When we have soft constraints and goals, it can be useful to give different priorities to them, and this should be taken into account in the plan quality evaluation. While there is more than one way to specify the importance of a soft constraint or goal, as a first attempt to tackle this issue, for IPC-5 we have chosen a simple quantitative approach: each soft constraint and goal is associated with a numerical weight representing the cost of its violation in a plan (and hence also its relative importance with respect the other specified soft constraints and goals). Weighted soft constraints and goals are part of the plan metric expression, and the best quality plans are those optimising such an expression (more details are given in the next sections).

Using this approach we can express that certain plans are more preferred than others. Some examples are (other formalised examples are given in the next sections):[1]

I prefer a plan where every airplane is used, rather than a plan using 100 units of fuel less, which could be expressed by weighting a failure to use all the planes by a number 100 times bigger than the weight associated with the fuel use in the plan metric; *I prefer a plan where each city is visited at most once, rather than a plan with a shorter makespan*, which could be expressed by using constraint violation costs penalising a failure to visit each city at most once very heavily; *I prefer a plan where at the end each truck is at its start location, rather than a plan where every city is visited by at most one truck*, which could be expressed by using goal costs penalising a goal failure of having every truck at its start location more heavily than a failure of having in the plan every city visited by at most one truck.

We also observe that the rich additional expressive power we propose to add for goal specifications allows the expression of constraints that are actually derivable necessary properties of optimal plans. By adding them as goal conditions, we have a way to express constraints that we know will lead to the planner finding optimal plans. Similarly, one can express constraints that prevent a planner from exploring parts of the plan space that are known to lead to inefficient performance.

In the next sections, we outline some extensions to PDDL2.2 that we propose for IPC-5. We call the extended language PDDL3.0. It should be noted that this is a preliminary version of the extended language, and that a more detailed description will be prepared in the future. Moreover, given that the proposed extensions are relatively new in the planning community, and that the teams participating in IPC-5 will have limited time to develop their systems, we impose some simplifying restrictions to make the language more accessible.

State Trajectory Constraints

Syntax and Intended Meaning

State trajectory constraints assert conditions that must be met by the entire sequence of states visited during the execution of a plan. They are expressed through temporal modal operators over first order formulae involving state predicates. We recognise that there would be value in also allowing propositions asserting the occurrence of action instances in a plan, rather than simply describing properties of the states visited during execution of the plan, but we choose to restrict ourselves to state predicates in this extension of the language. The use of the extensions described here imply a new requirements flag, `:constraints`.

The basic modal operators we propose to use in IPC-5 are: `always`, `sometime`, `at-most-once`, and `at end` (for goal state conditions). We use a special default assumption that unadorned conditions in the goal specification are automatically taken to be "at end" conditions. This assumption is made in order to preserve the standard meaning for existing goal specifications, despite the fact that in a standard semantics for an LTL formula an unadorned proposition would be interpreted according to the current state. We add `within` which can be used to express deadlines. In addition, rather than allowing arbitrary nesting of modal operators, we introduce some specific operators that offer some limited nesting. We have `sometime-before`, `sometime-after`, `always-within`. Other modalities could be added, but we believe that these are sufficiently powerful for an initial level of the sublanguage modelling constraints.

It should be noted that, by combining these modalities with timed initial literals (defined in PDDL2.2), we can express further goal constraints. In particular, one can specify the interval of time when a goal should hold, or the lower bound on the time when it should hold. Since these are interesting and useful constraints, we introduce two modal operators as "syntactic sugar" of the basic language: `hold-during` and `hold-after`.

Trajectory constraints are specified in the planning problem file in a new field, called `:constraints` that will usually appear after the goal. In addition, we allow constraints to be specified in the action domain file on the grounds that some constraints might be seen as safety conditions, or operating conditions, that are not physical limitations, but are nevertheless constraints that must always be respected in any valid plan for the domain (say legal constraints or operating procedures that must be respected). This also uses a section labelled `(:constraints ...)`. The interpretation of `(:constraints ...)` in the conjunction of a domain and a problem file is that it is equivalent to having all the constraints added to the goals. The use of trajectory constraints (in the domain file or in the goal specification) implies the need for the `:constraints` flag in the `:requirements` list.

Note that no temporal modal operator is allowed in preconditions of actions. That is, all action preconditions are with respect to a state (or time interval, in the case of `over all` action conditions).

The specific BNF grammar of PDDL3.0 is given in (Gerevini & Long 2005). The following is a fragment of the grammar concerning the new modalities of PDDL3.0 for expressing constraints (`con-GD`):

```
<con-GD> ::= (at end <GD>) | (always <GD>) |
             (sometime <GD>) | (within <num> <GD>) |
             (at-most-once <GD>) |
             (sometime-after <GD> <GD>) |
             (sometime-before <GD> <GD>) |
             (always-within <num> <GD> <GD>) |
             (hold-during <num> <num> <GD> |
             (hold-after <num> <GD> | ...
```

where `<GD>` is a goal description (a first order logic formula), `<num>` is any numeric literal (in STRIPS domains it will be restricted to integer values). There is a minor complication in the interpretation of the bound for `within` and `always-within` when considering STRIPS plans (and similarly for `hold-during` and `hold-after`): the question is whether the bound refers to sequential steps (in other words, actions) or to parallel steps. For STRIPS plans, the numeric bounds will be counted in terms of plan *happenings*. For

[1]The benchmark domains and problems of IPC-5 contain many additional examples; some samples of them are described in (Gerevini & Long 2006).

instance, `(within 10` ϕ`)` would mean that ϕ must hold within 10 happenings. These would be happenings of one action or of multiple actions, depending on whether the plan is sequential or parallel.

Notes on Semantics

The semantics of goal descriptors in PDDL2.2 evaluates them only in the context of a single state (the state of application for action preconditions or conditional effects and the final state for top level goals). In order to give meaning to temporal modalities, which assert properties of trajectories rather than individual states, it is necessary to extend the semantics to support interpretation with respect to a finite trajectory (as it is generated by a plan). We propose a semantics for the modal operators that is the same basic interpretation as is used in TLPlan (Bacchus & Kabanza 2000) for $\mathcal{LT}$ and other standard LTL treatments. Recall that a *happening* in a plan for a PDDL domain is the collection of all effects associated with the (start or end points of) actions that occur at the same time. This time is then the time of the happening and a happening can be "applied" to a state by simultaneously applying all effects in the happening (which is well defined because no pair of such effects may be mutex).

Definition 1 *Given a domain D, a plan π and an initial state I, π generates the trajectory*

$$\langle (S_0, 0), (S_1, t_1), ..., (S_n, t_n)\rangle$$

iff $S_0 = I$ and for each happening h generated by π, with h at time t, there is some i such that $t_i = t$ and S_i is the result of applying the happening h to S_{i-1}, and for every $j \in \{1 \dots n\}$ there is a happening in π at t_j.

Definition 2 *Given a domain D, a plan π, an initial state I, and a goal G, π is valid if the trajectory it generates, $\langle (S_0, 0), (S_1, t_1), ..., (S_n, t_n)\rangle$, satisfies the goal: $\langle (S_0, 0), (S_1, t_1), ..., (S_n, t_n)\rangle \models G$.*

This definition contrasts with the original semantics of goal satisfaction, where the requirement was that $S_n \models G$. The contrast reflects precisely this requirement that goals should now be interpreted with respect to an entire trajectory. We do not allow action preconditions to use modal operators and therefore their interpretation continues to be relative to the single state in which the action is applied. The interpretation of simple formulae, ϕ (containing no modalities), in a single state S continues to be as before and continues to be denoted $S \models \phi$. In the following definition we rely on context to make clear where we are using the interpretation of non-modal formulae in single states, and where we are interpreting modal formulae in trajectories.

Definition 3 *Let ϕ and ψ be atomic formulae over the predicates of the planning problem plus equality (between objects or numeric terms) and inequalities between numeric terms, and let t be any real constant value. The interpretation of the modal operators is as specified in Figure 1.*

Note that this interpretation exploits the fact that modal operators are not nested. A more general semantics for nested modalities is a straight-forward extension of this one. Note also that the last four expressions in Figure 1 are expressible in different ways if one allows nesting of modalities and use of the standard LTL modality until (more details on this in (Gerevini & Long 2005)).

The constraint `at-most-once` is satisfied if its argument becomes true and then stays true across multiple states and then (possibly) becomes false and stays false. Thus, there is only at most one *interval* in the plan over which the argument proposition is true.

For general formulae (which may or may not contain modalities):

$\langle (S_0, 0), (S_1, t_1), ..., (S_n, t_n)\rangle \models$ `(and` $\phi_1 ... \phi_n$`)` iff, for every i, $\langle (S_0, 0), (S_1, t_1), ..., (S_n, t_n)\rangle \models \phi_i$

and similarly for other connectives.

Of the constraints `hold-during` and `hold-after`, `(hold-during` t_1 t_2 ϕ`)` states that ϕ must be true during the interval $[t_1, t_2)$, while `(hold-after` t ϕ`)` states that ϕ must be true after time t. The first can be expressed by using timed initial literals to specify that a dummy timed literal `d` is true during the time window $[t_1, t_2)$ together with the goal `(always (implies d` ϕ`))`.
A variant of `hold-during` where ϕ must hold *exactly* during the specified interval could be easily obtained in a similar way. The second can be expressed by using timed initial literals to specify that `d` is true only from time t, together with the goal `(sometime-after d` ϕ`)`.

Soft Constraints and Preferences

A soft constraint is a condition on the trajectory generated by a plan that the user would prefer to see satisfied rather than not satisfied, but is prepared to accept might not be satisfied because of the cost of satisfying it, or because of conflicts with other constraints or goals. In case a user has multiple soft constraints, there is a need to determine which of the various constraints should take priority if there is a conflict between them or if it should prove costly to satisfy them. This could be expressed using a qualitative approach but, following careful deliberations, we have chosen to adopt a simple quantitative approach for this version of PDDL.

Syntax and Intended Meaning

The syntax for soft constraints falls into two parts. Firstly, there is the identification of the soft constraints, and secondly there is the description of how the satisfaction, or lack of it, of these constraints affects the quality of a plan.

Goal conditions, including action preconditions, can be labelled as preferences, meaning that they do not have to be true in order to achieve the corresponding goal or precondition. Thus, the semantics of these conditions is simple, as far as the correctness of plans is concerned: they are all trivially satisfied in any state. The role of these preferences is apparent when we consider the relative quality of different plans. In general, we consider plans better when they satisfy soft constraints and worse when they do not. A complication arises, however, when comparing two plans that satisfy different subsets of constraints (where neither set strictly contains the other). In this case, we rely on a specification of the violation costs associated with the preferences.

$$
\begin{array}{lll}
\langle (S_0,0),(S_1,t_1),...,(S_n,t_n)\rangle \models (\texttt{at end}\ \phi) & \text{iff} & S_n \models \phi \\
\langle (S_0,0),(S_1,t_1),...,(S_n,t_n)\rangle \models \phi & \text{iff} & S_n \models \phi \\
\langle (S_0,0),(S_1,t_1),...,(S_n,t_n)\rangle \models (\texttt{always}\ \phi) & \text{iff} & \forall i : 0 \leq i \leq n \cdot S_i \models \phi \\
\langle (S_0,0),(S_1,t_1),...,(S_n,t_n)\rangle \models (\texttt{sometime}\ \phi) & \text{iff} & \exists i : 0 \leq i \leq n \cdot S_j \models \phi \\
\langle (S_0,0),(S_1,t_1),...,(S_n,t_n)\rangle \models (\texttt{within}\ t\ \phi) & \text{iff} & \exists i : 0 \leq i \leq n \cdot S_i \models \phi \text{and } t_i \leq t \\
\langle (S_0,0),(S_1,t_1),...,(S_n,t_n)\rangle \models (\texttt{at-most-once}\ \phi) & \text{iff} & \forall i : 0 \leq i \leq n \cdot \text{ if } S_i \models \phi \text{ then} \\
 & & \exists j : j \geq i \cdot \forall k : i \leq k \leq j \cdot S_k \models \phi \\
 & & \quad \text{and } \forall k : k > j \cdot S_k \models \neg\phi \\
\langle (S_0,0),(S_1,t_1),...,(S_n,t_n)\rangle \models (\texttt{sometime-after}\ \phi\ \psi) & \text{iff} & \forall i \cdot \text{if } S_i \models \phi \text{ then } \exists j : i \leq j \leq n \cdot S_j \models \psi \\
\langle (S_0,0),(S_1,t_1),...,(S_n,t_n)\rangle \models (\texttt{sometime-before}\ \phi\ \psi) & \text{iff} & \forall i \cdot \text{if } S_i \models \phi \text{ then } \exists j : 0 \leq j < i \cdot S_j \models \psi \\
\langle (S_0,0),(S_1,t_1),...,(S_n,t_n)\rangle \models (\texttt{always-within}\ t\ \phi\ \psi) & \text{iff} & \forall i \cdot \text{if } S_i \models \phi \text{ then } \exists j : i \leq j \leq n \cdot S_j \models \psi \\
 & & \text{and } t_j - t_i \leq t
\end{array}
$$

Figure 1: Semantics of the basic modal operators in PDDL3.

The syntax for labelling preferences is simple:

```
(preference [name] <GD>).
```

The definition of a goal description can be extended to include preference expressions. However, in PDDL3.0, we reject as syntactically invalid any expression in which preferences appear nested inside any connectives, or modalities, other than conjunction and universal quantifiers. We also consider it a syntax violation if a preference appears in the condition of a conditional effect. *Note that where a named preference appears inside a universal quantifier, it is considered to be equivalent to a conjunction (over all legal instantiations of the quantified variable) of preferences all with the same name.*

Where a name is selected for a preference it can be used to refer to the preference in the construction of penalties for the violated constraint. The same name can be shared between preferences, in which case they share the same penalty.

Penalties for violation of preferences are calculated using the expression

```
(is-violated <name>)
```

where `<name>` is a name associated with one or more preferences. This expression takes on a value equal to the number of distinct preferences with the given name that are not satisfied in the plan. Note that in PDDL3.0 we do not attempt to distinguish degrees of satisfaction of a soft constraint — we are only concerned with whether or not the constraint is satisfied. Note, too, that the count includes each separate constraint with the same name. This means that:

```
(preference VisitParis
    (forall (?x - tourist)
            (sometime (at ?x Paris))))
```

yields a violation count of 1 for `(is-violated VisitParis)`, if at least one tourist fails to visit Paris during a plan, while

```
(forall (?x - tourist)
    (preference VisitParis
        (sometime (at ?x Paris))))
```

yields a violation count equal to the number of people who failed to visit Paris during the plan. The intention behind this is that each preference is considered to be a distinct preference, satisfied or not independently of other preferences. The naming of preferences is a convenience to allow different penalties to be associated with violation of different constraints.

Plans are awarded a value through the plan metric, introduced in PDDL2.1 (Fox & Long 2003). The constraints can be used in weighted expressions in a metric. For example,

```
(:metric minimize
     (+ (* 10 (fuel-used))
        (is-violated VisitParis)))
```

would weight fuel use as ten times more significant than violations of the `VisitParis` constraint. Note that the violation of a preference in the preconditions of an action is counted multiple times, depending on the number of the action occurrences in the plan. For instance, suppose that `p` is a preference in the precondition of an action a, which occurs three times in plan π. If the plan metric evaluating π contains the term `(* k (is-violated p))`, then this is interpreted as if it were `(* v (* k (is-violated p)))`, where `v` is the number of separate occurrences of a in π for which the preference is not satisfied.

Semantics

We say that

$$\langle (S_0,0),(S_1,t_1),...,(S_n,t_n)\rangle \models (\texttt{preference}\ \Phi)$$

is always true, so this allows preference statements to be combined in formulae expressing goals. The point in making the formula always true is that the preference is a soft constraint, so failure to satisfy it is not considered to falsify the goal formula. In the context of action preconditions, we say $S_i \models (\texttt{preference}\ \Phi)$ is always true, too, for the same reasons.

We also say that a preference `(preference` Φ`)` is *satisfied* iff $\langle (S_0,0),(S_1,t_1),...,(S_n,t_n)\rangle \models \Phi$ and *violated* otherwise. This means that `(or` Φ `(preference` Ψ`))` is the same as `(preference (or` $\Phi\,\Psi$`))`, both in terms of the satisfaction of the formulae and also in terms of whether the preference is satisfied. The same idea is applied to action precondition preferences. Hence, a goal such as:

```
(and (at package1 london)
      (preference (clean truck1)))
```

would lead to the following interpretation:

$\langle(S_0, 0), (S_1, t_1), ..., (S_n, t_n)\rangle \models$

```
    (and (at package1 london)
         (preference (clean truck1)
```

iff

$\langle(S_0, 0), (S_1, t_1), ..., (S_n, t_n)\rangle \models$

```
    (at package1 london)
```

and

$\langle(S_0, 0), (S_1, t_1), ..., (S_n, t_n)\rangle \models$

```
    (preference (clean truck1))
```

iff $S_n \models$ `(at package1 london)`

iff `(at package1 london)` $\in S_n$, since the preference is always interpreted as true. In addition, the preference would be *satisfied* iff:

$\langle(S_0, 0), (S_1, t_1), ..., (S_n, t_n)\rangle \models$

```
    (at end (clean truck1))
```

iff `(clean truck1)` $\in S_n$.

If the preference is not satisfied, it is violated.

Now suppose that we have the following preferences and plan metric:

```
(preference p1 (always (clean truck1)))
(preference p2 (and (at end (at package2 paris))
                    (sometime (clean track1))))
(preference p3 (...))

(:metric (+ (* 10 (is-violated p1)) (* 5 (is-violated p2))
            (is-violated p3))).
```

Suppose we have two plans, π_1, π_2, and π_1 does not satisfy preferences p1 and p3 (but it satisfies preference p2) and π_2 does not satisfy preferences p2 and p3 (but it satisfies preference p1), then the metric for π_1 would yield a value (11) that is higher than that for π_2 (6) and we would say that π_2 is better than π_1.

Formally, *a preference precondition is satisfied if the state in which the corresponding action is applied satisfies the preference*. Note that the restriction on where preferences may appear in precondition formulae and goals, together with the fact that they are banned from conditional effects, means that this definition is sufficient: the context of their appearance will never make it ambiguous whether it is necessary to determine the status of a preference. Similarly, *a goal preference is satisfied if the proposition it contains is satisfied in the final state*. Finally, *an invariant (*`over all`*) condition of a durative action is satisfied if the corresponding proposition is true throughout the duration of the action.*

In some case, it can be hard to combine preferences with an appropriate weighting to achieve the intended balance between soft constraints and other factors that contribute to the value of a plan (such as plan make span, resource consumption and so on). For example, to ensure that a constraint takes priority over a plan cost associated with resource consumption (such as make span or fuel consumption) is particularly tricky: a constraint must be weighted with a value that is higher than any possible consumption cost and this might not be possible to determine. With non-linear functions it is possible to achieve a bounded behaviour for costs associated with resources. For example, if a constraint, C, is to be considered always to have greater importance than the make span for the plan then a metric could be defined as follows:

```
(:metric minimize (+ (is-violated C)
                     (- 1 (/ 1 (total-time))))).
```

This metric will always prefer a plan that satisfies C, but will use make span to break ties.

Nevertheless, for the competition, where it is important to provide an unambiguous specification by which to rank plans, the use of plan metrics in this way is clearly very straightforward and convenient. We leave for later proposals the possibilities for extending the evaluation of plans in the face of soft constraints.

Some Examples

The following state trajectory constraints could be stated either as strong constraints or soft constraints.

"A fragile block can never have something above it":

```
(always (forall (?b - block)
          (implies (fragile ?b) (clear ?b))))
```

"A fragile block can have at most one block on it":

```
(always (forall (?b1 ?b2 - block)
          (implies (and (fragile ?b1) (on ?b2 ?b1))
                   (clear ?b2))))
```

"The blocks forming the same tower always have the same color":

```
(always (forall (?b1 ?b2 - block ?c1 ?c2 - color)
          (implies (and (on ?b1 ?b2) (color ?b1 ?c1)
                        (color ?b2 ?c2))
                   (= ?c1 ?c2))))
```

"Each block should be picked up *at least* once":

```
(forall (?b - block) (sometime (holding ?b)))
```

"Each block should be picked up *at most* once":

```
(forall (?b - block) (at-most-once (holding ?b)))
```

"In some state visited by the plan all blocks should be on the table":

```
(sometime (forall (?b - block) (on-table ?b)))
```

This constraint requires all the blocks to be on the table in the *same* state. In contrast, if we only require that every block should be on the table in *some* state we can write:

```
(forall (?b - block) (sometime (on-table ?b)))
```

"Whenever I am at a restaurant, I want to have a reservation":

```
(always (forall (?r - restaurant)
        (implies (at ?r) (have-reservation ?r))))
```

"Each truck should visit each city *at most* once":

```
(forall (?t - truck ?c - city) (at-most-once (at ?t ?c)))
```

"At some point in the plan all the trucks should be at city1":

```
(sometime (forall (?t - truck) (at ?t city1)))
```

"Each truck should visit each city *exactly once*":

```
(and (forall (?t - truck ?c - city)
             (at-most-once (at ?t ?c)))
     (forall (?t - truck ?c - city)
             (sometime (at ?t ?c))))
```

"Each city is visited by at most one truck at the same time":

```
(forall (?t1 ?t2 - truck ?c1 city)
   (always (implies (and (at ?t1 ?c1)
                    (at ?t2 ?c1)) (= ?t1 ?t2))))
```

The following two examples use the IPC-3 Rovers domain involving numerical fluents. "We would like that the energy of every rover should always be above the threshold of 5 units":

```
(always (forall (?r - rover) (> (energy ?r) 5))))
```

"Whenever the energy of a rover is below 5, it should be at the recharging location within 10 time units":

```
(forall (?r - rover)
   (always-within 10 (< (energy ?r) 5)
                     (at ?r recharging-point)))
```

The next two examples illustrate the usefulness of `sometime-before` and `sometime-after`. The first one states that "a truck can visit a certain city (where initially there is no truck) only after having visited another particular one"; the second one that "if a taxi has been used and it is at the depot, then it has to be cleaned" (if a taxi is used but it does not go back to the depots, then there is no need to clean it).

```
(forall (?t - truck)
    (sometime-before (at ?t city1) (at ?t city2)))

(forall (?t - taxi)
    (sometime-after (and (at ?t depot) (used ?t))
                    (clean ?t)))
```

"We want a plan moving package1 to London such that truck1 is always maintained clean, and at some point truck2 is at Paris. Moreover, we also prefer that truck3 is always clean and that at the end of the plan package2 is at London":

```
(:goal (and (at package1 london)
            (preference (at package2 london))))

(:constraints
   (and (always (clean truck1))
        (sometime (at truck2 paris))
        (preference (always (clean truck3)))
        (preference (at end (at package2 london)))))
```

"We prefer that every fragile package to be transported is insured".

```
(forall (?p - package)
    (preference P1
        (always (implies (fragile ?) (insured ?p)))))
```

We now consider an example with a plan metric.
"We want three jobs completed. We would prefer to take a coffee-break and that we take it when everyone else takes it (at coffee-time) rather than at any time. We would also like to finish reviewing a paper, but it is less important than taking a break. Finally, we would like to be finished so that we can get home at a reasonable time, and this matters more than finishing the review or having a sociable coffee break":

```
(:goal (and (finished job1)
            (finished job2)
            (finished job3)) )

(:constraints
  (and (preference break
         (sometime (at coffee-room)))
       (preference social
         (sometime (and (at coffee-room) (coffee-time))))
       (preference reviewing (reviewed paper1))))

(:plan-metric minimize
               (+  (* 5 (total-time))
                   (* 4 (is-violated social))
                   (* 2 (is-violated break))
                   (is-violated reviewing)))
```

Now consider three plans, π_1, π_2 and π_3, such that all three plans complete the three jobs. Suppose π_1 achieves this in 4 hours, but takes no break and does not include reviewing the paper. Suppose π_2 completes the jobs in 8 hours, but takes a coffee-break at coffee-time and reviews the paper. Finally, π_3 completes the jobs in 6 hours, including reviewing the paper, but only by taking a short break when the coffee room is empty. Then the values of the plans are:

Plan	Quality
π_1	5*4 + 4*1 + 2*1 + 1 = 27
π_2	5*8 + 4*0 + 2*0 + 0 = 40
π_3	5*6 + 4*1 + 2*0 + 0 = 34

This makes π_1 the best plan and π_2 the worst.

Plan Validation and Evaluation

A plan validator will be developed as an extension of the existing validator used in the previous competitions. The two key aspects of this extension are checking state trajectory constraints in the goal, which does not complicate the execution simulation for a plan, and the checking of preferences in order to compare plans. This latter extension will involve identifying the constraint violations associated with each plan and their violation times, in order to evaluate the plan quality according to the specified metric (which may include terms for the preference violations). The organizers of IPC-5 are considering the possibility of using different variants of the test problems involving only strong constraints or soft constraints, with a possible additional distinction between simple preferences, involving only goals or action preconditions, and more complex preferences involving general soft constraints. More details about this organization of the benchmarks will be announced in the the web page of the deterministic track of IPC-5: `http://ipc5.ing.unibs.it`.

Extensions and Generalization

There is considerable scope for developing the proposed extension. First, and most obviously, modal operators could be allowed to nest. This would allow a rich expressive power in the specification of modal temporal goals. Nesting would allow constraints to be applied to parts of trajectories, as is usual in modal temporal logics. In addition, we could introduce propositions representing that an action appears in a plan.

Other modal operators could be added. We have excluded them PDDL3.0 because we have found that many interesting and challenging goals can be captured without them,

$$\langle (S_0, 0), (S_1, t_1), ..., (S_n, t_n)\rangle \models (\texttt{always-persist}\ t\ \phi) \quad \text{iff} \quad \forall i : 0 < i \leq n \cdot \text{if } S_i \models \phi \text{ and } S_{i-1} \models \neg\phi \text{ then } \exists j : j - i \geq t \cdot \forall z : i \leq z \leq j \cdot S_z \models \phi \text{ and if } S_0 \models \phi \text{ then } \forall z : z \leq t \cdot S_z \models \phi$$

$$\langle (S_0, 0), (S_1, t_1), ..., (S_n, t_n)\rangle \models (\texttt{always-persist}\ t\ \phi) \quad \text{iff} \quad \exists i : 0 < i \leq n \cdot \text{if } S_i \models \phi \text{ and } S_{i-1} \models \neg\phi \text{ then } \exists j : j - i \geq t \cdot \forall z : i \leq z \leq j \cdot S_z \models \phi, \text{ or if } S_0 \models \phi \text{ then } \forall z : z \leq t \cdot S_z \models \phi$$

Figure 2: Semantics of always-persist and sometime-persist.

and we do not wish to add unnecessarily to the load on potential competitors. The modal operator until would be an obvious one to add. Without nesting, a related always-until and sometime-until would allow expression of goals such as "every time a truck arrives at the depot, it must stay there until loaded" or "when the truck arrives at the depot, it must stay there until cleaned and fully refuelled at least once in the plan". The formal semantics of always-until and sometime-until can be easily derived from the one of until in LTL. By combining always-until and other modalities we can express complex constraints such as that "whenever the energy of a rover is below 5, it should be at the recharging location within 10 time units and remain there until recharged":

```
(and (always-until (charged ?r) (at ?r rechargepoint))
     (always-within 10 (< (charge ?r) 5)
                       (at ?r rechargingpoint)))
```

Another modality that would be an useful extension of the expressive power is a complement for `within`, such as persist, with the semantics that a proposition once made true must persist for at least some minimal period of time. Without nesting, a related always-persist and sometime-persist would allow expression of goals such as "I want to spend at least 2 days in each of the cities on my tour", or "every time the taxi goes to the station it must wait for at least 10 without a passenger".
The formal semantics of always-persist and sometime-persist is given in Figure 2. A generalisation that would allow `within` and persist to be combined would be to allow the time specification to be associated with a comparison operator to indicate whether the bound is an upper or lower bound.

We have deliberately not introduced the operator next, which is common in modal temporal logics. This is because concurrent fragments of a plan might cause a state change that is not relevant to the part of the state in which the next condition is intended to apply. Furthermore, the fact that PDDL plans are embedded on a real time line means that the intention behind next is less obviously relevant. We realise that next has been particularly useful in expressing control rules for planners like TALPlanner (Kvarnström & Magnusson 2003) and TLPlan (Bacchus & Kabanza 2000), but our intention in developing this extension is to focus on providing a language that is useful for expressing constraints that govern plan quality, rather than for control knowledge. We believe that the use of `always-within` captures a much more useful concept for plan quality that is actually a far more realistic constraint in modelling planning problems.

Extensions to the use of soft constraints include the definition of more complex preferences, such as conditional preferences, and a possible qualitative method for expressing priorities over preferences. Moreover, the evaluation of the soft constraints could be extended by considering a degree of constraint violation, such as the amount of time when an `always` constraint is violated, the delay that falsifies a `within` constraint, or the number of times an `always-after` constraint is violated.

Acknowledgments

We would like to thank Y. Dimopoulos, C. Domshlak, S. Edelkamp, M. Fox, P. Haslum, J. Hoffmann, A. Jonsson, D. McDermott, A. Saetti, L. Schubert, I.Serina, D. Smith and D. Weld for some very useful discussions about PDDL3.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

Brafman, R., and Chernyavsky, Y. 2005. Planning with goal preferences and constraints. In *Proc. of ICAPS-05*.

Briel, M.; Sanchez, R.; Do, M.; and Kambhampati, S. 2004. Effective approaches for partial satisfaction (over-subscription) planning. In *Proc. of the AAAI-04*.

Delgrande, P. J.; Schaub, T.; and Tompits, H. 2005. A general framework for expressing preferences in causal reasoning and planning. In *Proc. of the 7^{th} Int. Symposium on Logical Formalizations of Commonsense Reasoning*.

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

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3. Technical Report RT-2005-08-47, Dep. di Elettronica per l'Automazione, Universitá di Brescia, Italy. An extension with the BNF grammar of PDDL3.0 is available from `http://ipc5.ing.unibs.it`.

Gerevini, A., and Long, D. 2006. Preferences and soft constraints in PDDL3. In *Proc. of ICAPS Workshop on Preferences and Soft constraints in Planning*.

Kvarnström, J., and Magnusson, M. 2003. Talplanner in the 3rd international planning competition: Extensions and control rules. *Journal of AI Research* 20.

Miguel, I.; Jarvis, P.; and Shen, Q. 2001. Efficient flexible planning via dynamic flexible constraint satisfaction. *Engineering Applications of Artificial Intelligence* 14(3):301–327.

Smith, D. 2004. Choosing objectives in over-subscription planning. In *Proc. of ICAPS-04*.

Son, T., C., and Pontelli, E. 2004. Planning with preferences using logic programming. In *Proc. of LPNMR-04*. Springer-Verlag. LNAI 2923.

The Benchmark Domains of the Deterministic Part of IPC-5

Yannis Dimopoulos$^+$ **Alfonso Gerevini*** **Patrik Haslum**$^\circ$ **Alessandro Saetti***
$^+$ Department of Computer Science, University of Cyprus, Nicosia, Cyprus
* Department of Electronics for Automation, University of Brescia, Brescia, Italy
$^\circ$ Department of Computer and Information Science, Linköping University, Linköping, Sweden
$^+$yannis@cs.ucy.ac.cy *{gerevini,saetti}@ing.unibs.it $^\circ$pahas@ida.liu.se

Abstract

We present a set of planning domains and problems that have been used as benchmarks for the fifth International planning competition. Some of them were inspired by different types of logistics applications, others were obtained by encoding known problems from operation research and bioinformatics. For each domain, we developed several variants using different fragments of PDDL3 with increasing expressiveness.

Introduction

The language of the fifth International planning competition (IPC-5), PDDL3.0 (Gerevini & Long 2005), is an extension of the previous versions of PDDL (Fox & Long 2003; Edelkamp & Hoffmann 2004) that aims at a better characterization of plan quality. The new language allows us to express *strong and soft constraints on plan trajectories* (i.e., constraints over intermediate states reached by the plan), as well as *strong and soft problem goals*. Strong trajectory constraints and goals must be satisfied by any valid plan, while soft trajectory constraints and goals (called *preferences*) express desired constraints and goals, which do not necessarily have to be achieved. In PDDL3.0, the plan metric expression can include weighted penalty terms associated with the violation of the soft trajectory constraints and goals in the problem.

This paper gives an informal presentation of the benchmark domains and problems that we developed for IPC-5, and that include most of the new features of PDDL3.0.[1] We designed five new domains, as well as some new variants of two domains that have been used in previous planning competitions. In order to make the language more accessible to the the IPC-5 competitors, we developed for each domain several variants, using different fragments of PDDL3.0. The "propositional" and "metric-time" variants use only the constructs of PDDL2.2 (Edelkamp & Hoffmann 2004); the "simple preferences" variant extends the propositional with preferences over the problem goals; the "qualitative preferences" variant also includes preferences over state trajectory constraints; the "metric-time constraints" variant extends the metric-time variant with strong state trajectory constraints; and, finally, the "complex preferences" variant uses the full power of the language, including soft trajectory constraints and goals. However, not all the different variants of each domain actually use the full fragment "allowed" for that variant.

In the domain variants involving preferences we created for each planning problem a plan metric incorporating terms specifying the penalties for violations of the preference. The metric is a very important part of the problem statements in such domains, since it determines which is the best trade-off between different, perhaps mutually exclusive, preferences, and we tried with much care to ensure that the metrics in the test problems give rise to challenging optimization problems.

The IPC-5 test domains have different motivations. Some of them were inspired by real world applications, (e.g., `storage`, `trucks` and `pathways`); others were aimed at exploring the applicability and effectiveness of automated planning for new applications (`pathways`), or for known problems that have been addressed in other fields of computer science (`TPP` and `openstacks`); finally, two domains were taken from previous competitions, as sample references for the advancement of automated planning with respect to the existing benchmarks (`rovers` and `pipesworld`).

For some domains, the problems we generated have many solutions. In these problems, the most challenging aspect is finding plans of good quality. Other problems are challenging for different reasons: the expressiveness of the planning language used to model the problem including some of the new features of PDDL3.0, the large size of the problem, or the known NP-hardness of the computational problem they model. In most cases, the test problems were automatically (or semi-automatically) generated by using dedicated software tools.

[1]A detailed description of the IPC-5 benchmarks is outside the scope of this short paper; their PDDL formalization is available from the IPC-5 website: `http://ipc5.ing.unibs.it`.

The Travelling Purchaser Domain

This is a relatively recent planning domain that has been investigated in operations research (OR) for several years, e.g., (Riera-Ledesma & Salazar-Gonzalez 2005). The Travelling Purchaser Problem (TPP) is a known generalization of the Travelling Salesman Problem, and is defined as follows. We have a set of products and a set of markets. Each market can provide a limited amount of each product at a known price. The TPP consists in selecting a subset of markets such that a given demand for each product can be purchased, minimizing the combined travel and purchase cost. This problem arises in several applications, mainly in routing and scheduling contexts, and it is NP-hard. In OR, computing optimal or near optimal solutions for the TPP instances is still an active research topic.

For IPC-5, we have formalized several variants of this domain in PDDL. One of them is equivalent to the original TPP, while the others are different formulations or significant (we believe and hope) extensions. In all these domain variants, plan quality is important, although for some instances even finding an arbitrary solution could be quite difficult for a fully-automated planner.

For this domain, we developed both a metric version without time and a metric-time version. We begin the description with the metric version because it is the one equivalent to the original formulation of the TPP.

Metric

This version is equivalent to the original formulation of the TPP in OR. There are only three operators, two of which are used to model the purchasing actions: "buy-all" and "buy-allneeded". The first buys at a certain market (`?m`) the whole amount of a type of goods (`?g`) sold by the market (`?m` and `?g` are operator parameters); while the second one buys at `?m` the amount of `?g` that is needed to complete the purchase of `?g` (as specified in the problem goals). In this version, every market is *directly* connected to every other market and to the depots. Moreover, there is only one depot and only one truck.

Propositional

This version models a variant of the original TPP where: (1) there can be more than one depot and more than one truck; (2) the amount of goods are discrete and represented by qualitative levels; (3) every type of goods has the same price, independent from the market where we buy it; (4) there are two new operators for loading and unloading goods to/from trucks; (5) markets and depots can be indirectly connected.

Simple Preferences

The operators in this domain are the same as in the propositional version. The difference is in the goals, which are all soft goals (preferences). These preferences concern maximizing the level of goods that are stored in the depots, constraints between the levels of different stored goods, and the safety condition that all purchased goods are stored at some market.

Qualitative Preferences

The operators in this version are the same as in the propositional version. All goals are preferences concerning maximizing, for every type of goods, the purchased and stored levels. This version includes preferences over trajectory constraints. These are constraints between the levels of two types of stored goods; constraints about the use of the trucks for loading goods; constraints imposing the use of every truck. Moreover, we have the preference that in the final state all purchased goods are stored at some depot.

Metric-Time

With respect to the simpler metric version, which is equivalent to the original formulation of the TPP, this version has the the following main differences: same as points (1), (4), (5) illustrated in the description of the propositional variants; each action has a duration and the plan quality is a linear combination of total-time (makespan) and the total cost of traveling and purchasing; the operator "`buyall`" has a "`rebate`" rate (if you buy the whole amount of a type of goods that is sold at a market, then you have a discount).

Metric-Time Constraints

The operators in this version are the same as in the metric-time version. In addition, in the domain file, we have some strong constraints imposing that in the final state all purchased goods are stored, every market can be visited by at most one truck at the same time, every truck is used. Moreover, in the problem specification, we have several strong constraints about the relative amounts of different types of goods stored in a depot, the number of times a truck can visit a market, the order in which goods should be stored, the order in which we should store some type of goods and buy another one, and deadlines about delivering goods once they have been loaded in a truck.

Complex Preferences

The operators in this version are the same as in the metric-time version. In addition, it contains many preferences over state trajectory constraints that are similar to those used for the metric-time constraints version.

The Openstacks Domain

The openstacks domain is based on the "minimum maximum simultaneous open stacks" combinatorial optimization problem, which can be stated as follows:

A manufacturer has a number of orders, each for a combination of different products, and can only make one product at a time. The total required quantity of each product is made at the same time (because changing from making one product to making another requires a production stop). From the time that the first product included in an order is made to the time that all products included in the order have been made, the order is said to be "open" and during this time it requires a "stack" (a temporary storage space). The problem is to order the making of the different products so that the maximum number of stacks that are in use simultaneously, or equivalently the number of orders that are in simultaneous production, is minimized (because each stack takes up space in the production area).

This problem, and many related variants, have been studied in operations research (see, e.g., Fink & Voss 1999). It is known to be NP-hard, and equivalent to several other problems (Linhares & Yanasse 2002). This is a pure optimization problem: for any instance of the problem, every ordering of the making of products is a solution, which at worst uses as many simultaneously open stacks as there are orders. Thus, finding a plan is quite trivial (in the sense that there exists a domain-specific linear-time algorithm that solves the problem), but finding a plan of high quality is hard (even for a domain-specific algorithm).

The openstacks problem was recently posed as a challenge problem for the constraint programming community, and, as a result, a large library of problem instances, together with results on those instances for a number of different solution approaches, are available (see Smith & Gent (2005)).

Propositional

This variant is simply an encoding of the original openstacks problem as a planning problem. The encoding is done in such a way that minimizing the length (sequential or parallel) of the plan also minimizes the objective function, i.e., the maximum number of simultaneously open stacks. There are three basic actions to start orders, make products, and ship orders once they are completed, plus an action that "opens" a new stack, but in order to ensure the correspondance between parallel length and the objective function, some of these actions are split in two parts. The domain formulation uses some ADL constructs (quantified disjunctive preconditions), but these can be compiled away with only a linear increase in size.

The problems are a selection of the problems used in the constraint modelling challenge, including a few problems that could not be solved (optimally) by any of the CSP approaches, plus a small number of extra small instances.

Time

In this variant of the domain the number of available stacks is fixed, and the objective is instead to minimize makespan. Makespan is dominated by the actions that make products. The number of stacks is for each problem chosen to be somewhere between the optimal and the trivial upper bound (equal to the number of orders).

Metric-Time

In this variant, the objective function is to minimize a (linear) combination of the number of open stacks and the plan makespan. The number of open stacks is modelled using numeric fluents.

Simple Preferences

In this variant, the goal of including all required products in each order is softened, and a "score" (or "reward") is instead given for each product that is included in an order when it is shipped. The objective is to maximize this score. The maximum number of open stacks is fixed, like in the temporal variant, but at a number slightly less than the optimal number required to satisfy all the requirements of all orders.

This version of the domain uses an ADL construct (a quantified conditional effects) that can only be compiled away at an exponential increase in problem size.

Complex Preferences

This version, like the previous, has soft goals, but also a variable maximum number of open stacks. The objective is to maximize a linear combination of the score (positive) and the number of open stacks (negative). Also like the previous version, the formulation uses a quantified conditional effect.

The Storage Domain

"Storage" is a planning domain involving spatial reasoning. Basically, the domain is about moving a certain number of crates from some containers to some depots by hoists. Inside a depot, each hoist can move according to a specified spatial map connecting different areas of the depot. The test problems for this domain involve different numbers of depots, hoists, crates, containers, and depot areas. While in this domain it is important to generate plans of good quality, for many test problems, even finding any solution can be quite hard for domain-independent planners.

Altogether, the different variants of this domain, involve almost all the new features of PDDL3.0. Note that this domain is basically a propositional domain, where the space for storing crates is represented by PDDL literals. For this domain, instead of a metric-time version, we have a "time-only" version (without numerical fluents).

Propositional

The domain has five different actions: an action for lifting a crate by a hoist, an action for dropping a crate by a hoist, an action for moving a hoist into a depot, an action for moving a hoist from one area of a depot to another one, and finally an action for moving a hoist outside a depot.

Time

This variant is basically the propositional variant where the actions have duration and the plan quality is total-time (plan makespan).

Simple Preference

The operators in this domain are the same as those in the propositional version. The main difference is in the goals. All goals are soft goals (preferences). These preferences concern which depots and depot areas should be used for storing the crates, the desire that only "compatible" crates are stored in the same depot, the desire that the incompatible crates stored in the same depot are located at non-adjacent areas of the depot and, finally, the desire that the hoists are located in depots different from those where we store the crates.

Qualitative Preferences

The operators in this domain are the same as those in the propositional version. The differences are in the preferences over the goals and state trajectory constraints. All goals are soft goals similar to some of the soft goals specified in the simple preferences variant. The preferences over trajectory constraints concern constraints about the use of the available hoists for moving the crates, and about the order in which crates are stored in the depots. Moreover, we have the preference that in any state crossed by the plan, the adjacent areas in a depot can be occupied only by compatible crates.

Time Constraints

The operators in this version are the same as those in the temporal version. The problem goals are specified by an "at-end" constraint imposing that all crates are stored in a depot. The problems have several constraints imposing that a crate can be lifted at most once, ordering constraints about storing certain crates before others, deadlines for storing the crates, and maximum time a hoist can stay outside a depot. There are also constraints imposing a safety condition, that in the final state, all hoists are inside a depot; some constraints imposing that every hoist is used; and some constraints imposing that incompatible crates are not stored at adjacent areas of the depot.

Time Preferences

The operators in this version are the same as those in the temporal version. In addition, this version contains many preferences over state trajectory constraints that are similar to those used for the time constraints version.

The Trucks Domain

Essentially, this is a logistics domain about moving packages between locations by trucks under certain constraints. The loading space of each truck is organized by areas: a package can be (un)loaded onto an area of a truck only if the areas between the area under consideration and the truck door are free. Moreover, some packages must be delivered within a deadline. In this domain, it is important to find good quality plans. However, for many test problems, even finding one plan could be a rather difficult task.

Like the Storage domain, this domain has a "time-only" variant instead of a metric-time variant (i.e., there are no numerical fluents). The other variants make extensive use of the new features of PDDL3.0. We start the description from the time constraint version, because it is the one closest to a realistic problem.

Time Constraints

The domain has four different actions: an action for loading a package into a truck, one for unloading a package from a truck, one for moving a truck, and finally one for delivering a package. The durations of loading, unloading and delivering packages are negligible compared to the durations of the driving actions. The problem goals require that certain packages are at their final destinations by certain deadlines. For this variant, we also created an equivalent version, "Time-TIL", in which the trajectory constraints of type "`within`" are compiled into timed initial literals. Each competing team is free to choose one of the two alternative variants.

Time

The operators are the same as those in the time constraints version, but there is no deadline for delivering packages. Finding a valid plan in this version is significantly easier, but finding a plan with short makespan is still challenging.

Complex Preferences

The operators in this version are the same as those in the constraints version. The deadlines are modeled by preferences. Moreover, this version contains preferences over trajectory constraints. These are constraints imposing some ordering about when delivering packages, constraints about the usage of the areas in the trucks, and constraints about loading packages.

Propositional

The operators in this version are similar to those in the constraints version, with the main difference that time is modeled as a discrete resource (with a fixed number of levels). Moreover, the driving actions cannot be executed concurrently.

Simple Preferences

The operators in this domain are the same as those in the propositional version. The difference concerns the problem goals where the delivering deadlines are modeled by preferences.

Qualitative Preferences

The operators in this domain are the same as those in the propositional version. The difference concerns the problems goals including soft delivering deadlines. Moreover, this version includes many preferences over state trajectory constraints that are similar to those used for the complex preferences version.

The Pathways Domain

This domain is inspired by the field of molecular biology, specifically biochemical pathways. “A pathway is a sequence of chemical reactions in a biological organism. Such pathways specify mechanisms that explain how cells carry out their major functions by means of molecules and reactions that produce regular changes. Many diseases can be explained by defects in pathways, and new treatments often involve finding drugs that correct those defects.” (Thagard 2003) We can model parts of the functioning of a pathway as a planning problem by simply representing chemical reactions as actions. The goal in these planning problems is to construct a sequence of reactions that produces one or more substances, using a limited number of substances as input. The planner is partly free to choose which input substances to use, i.e., to choose some aspects of the initial state of the problem. This aspect of the problem is modelled by means of additional actions.

The biochemical pathway domain of the competition is based on the pathway of the Mammalian Cell Cycle Control as it described in (Kohn 1999) and modelled in (Chabrier 2003). There are three different kinds of basic actions corresponding to the different kinds of reactions that can appear in a pathway.

Propositional

This is a simple qualitative encoding of the reactions of the pathway. The domain has five different actions: an action for choosing the initial substances, an action for increasing the quantity of a chosen substance (in the propositional version, quantity coincides with presence, and it is modeled through a predicate indicating if a substance is available or not), an action modeling biochemical association reactions, an action modeling biochemical association reactions requiring catalysts, and an action modeling biochemical synthesis reactions. Also, there is an additional set of “dummy” actions used to encode the disjunctive problem goals.

The goals refer to substances that must be synthesized by the pathway, and are disjunctive with two disjuncts each. Furthermore, there is a limit on the number of input substances that can be used by the pathway.

Simple Preferences

This is similar to the propositional version, with the difference that both the products that must be synthesized by the pathway and the number of the input reactants that are used by the network are turned into preferences. The challenge here is finding plans that achieve a good tradeoff between the different kinds of preferences.

Metric-Time

In this version of the domain, reactions have different durations. The reactions can only happen if their input reactants reach some concentration level, and reactions generate their products in specific quantities. The goals in this version are summations of substance concentrations that must be generated by the reactions of the pathway. The plan metric minimizes some linear combination of the number of input substances and the plan duration.

Complex Preferences

This is an extension of the metric-time version with different preferences concerning the concentration of substances of the pathway, or the order in which substances are produced. The metric is a combination of these preferences, the number of substances used and the plan makespan.

The Extended Rovers Domain

The Rovers domain was introduced in the 2002 planning competition (Long & Fox 2003). It models the problem of planning for a group of planetary rovers to explore the planet they are on (taking pictures and samples from interesting locations).

Propositional and Metric-Time

The propositional and metric-time versions of the domain are the same as in IPC 2002, with the addition of some planning problems.

The domain has nine different actions: an action for moving rovers on a planet surface, two actions for sampling soil and rock, an action for dropping rock or soil, an action for calibrating rover instruments, an action for taking image of interesting objective, and finally three actions for transmitting soil data, rock data or image data.

Qualitative Preferences

This is the IPC 2002 propositional version with soft trajectory constraints added (constraint types always, sometime and at-most-once are used). The objective is simply to maximize the number of preferences satisfied. The preferences are “artificial”, in the sense that they do not encode any “real” preferences on the plan, but are constructed in a way as to make the problem of maximizing the satisfaction of preferences challenging.

Metric Simple Preferences

This version is a special case of the complex preferences version, which has preferences only on the goals of the problem.

This version of the domain poses a so-called "net benefit" problem: goals (atoms, and in some cases conjunction of atoms) have values and actions have cost, and the objective is to maximize the sum values of achieved goals minus the sum of costs of actions in the plan. Only the actions that move the rovers have non-zero cost. The domain uses simple (goal state) preferences to encode goal values and fluents to encode action costs. There are three different sets of problems, with somewhat different properties. In the first, goals are *interfering*, meaning that the cost of achieving any two goals is greater than the sum of achieving them individually. The second has instead *synergy* between the goals, i.e., the cost of achieving several goals is less than the sum of achieving each of them separately, while the third contains goals with relationships of both kinds.

The Extended Pipesworld Domain

The Pipesworld domain was introduced in the previous planning competition (Hoffmann & Edelkamp 2005). It models the transportation of batches of petroleum products in a network of pipelines.

Propositional and Time

The propositional and temporal versions of the domain are the "tankage" variant of the domain used in IPC 2004 The domain has six actions: two actions for moving a batch from a tankage to a pipeline segment (one for the start and one for the end of the activity), two actions for moving a batch from a tankage to a pipeline segment, and two actions for moving a batch from a tankage (or pipeline segment) to a pipeline segment (or tankage) in case the pipes consist of only one segment.

Time Constraints

The time constraints variant is based on the temporal no-tankage variant from IPC 2004, but adds hard deadlines on when each of the goals must be reached. Deadlines are specified using the PDDL3 `within` constraint. The problems also have a number of "triggered" deadline constraints, specified with PDDL3 `always-within` constraint.

Complex Preferences

This variant is similar to the previous, but has soft deadlines instead, encoded with preferences on the constraints. Each goal can have several (increasing) deadline, with different (increasing) penalties for missing them.

Conclusions

We have given an informal description of the benchmark domains that we developed for the deterministic part of the 2006 International Planning Competition. The general aim was to create a new set of problems for the planning community involving new and interesting – and hopefully also useful – issues, in particular planning with (possibly contradicting) preferences over problem goals and state trajectory constraints.

Several competing teams have declared their that their planners are capable of handling parts of the extended PDDL3 language. At the time of writing, benchmark tests are still being run. In addition to their use for the competition, we hope that the new benchmarks will provide a challenging extension to the existing set of planning benchmarks, both those involving PDDL3 constructs and those that can be specified through the previous versions of PDDL.

References

Chabrier, N. 2003. `http://contraintes.inria.fr/BIOCHAM/EXAMPLES/~cell_cycle/cell_cycle.bc`.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classic part of the 4th international planning competition. Technical Report 195, Institut für Informatik, Freiburg, Germany.

Fink, A., and Voss, S. 1999. Applications of modern heuristic search methods to pattern sequencing problems. *Computers & Operations Research* 26:17 – 34.

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

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3. Technical report rt-2005-08-47, Universitá di Brescia, Dipartimento di Elettronica per l'Automazione.

Hoffmann, J., and Edelkamp, S. 2005. The deterministic part of IPC-4: An overview. *Journal of AI Research* 24:519 – 579.

Kohn, K. 1999. Molecular interaction map of the mammalian cell cycle control and dna repair systems. *Mol Biol Cell* 10(8).

Linhares, A., and Yanasse, H. 2002. Connection between cutting-pattern sequencing, VLSI design and flexible machines. *Computers & Operations Research* 29:1759 – 1772.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20:1 – 59.

Riera-Ledesma, J., and Salazar-Gonzalez, J., J. 2005. A heuristic approach for the travelling purchaser problem. *European Journal of Operational Research* 160(3):599–613.

Smith, B., and Gent, I. 2005. Constraint modelling challenge 2005. `http://www.dcs.st-and.ac.uk/~ipg/challenge/`.

Thagard, P. 2003. Pathways to biomedical discovery. *Philosophy of Science* 70.

Planning with Temporally Extended Preferences by Heuristic Search

Jorge Baier and **Jeremy Hussell** and **Fahiem Bacchus** and **Sheila McIlraith**
Department of Computer Science
University of Toronto
Toronto, Canada
[jabaier|hussell|fbacchus|sheila]@cs.toronto.edu

Abstract

In this paper we describe a planner that extends the TLPLAN system to enable planning with temporally extended preferences specified in PDDL3, a variant of PDDL that includes descriptions of temporal plan preferences. We do so by compiling preferences into nondeterministic finite state automata whose accepting conditions denote achievement of the preference described by the automaton. Automata are represented in the planning problem through additional predicates and actions. With this compilation in hand, we are able to use domain-independent heuristics to guide TLPLAN towards plans that realize the preferences. We are entering our planner in the *qualitative preferences* track of IPC5, the 2006 International Planning Competition. As such, the planner description provided in this paper is preliminary pending final adjustments in the coming weeks.

Introduction

Standard goals in planning allow us to distinguish between plans that satisfy the goal and those that do not, however, they fail to discriminate between the quality of different successful plans. Preferences, on the other hand, express information about how "good" a plan is thus allowing us to distinguish between desirable successful plans and less desirable successful plans.

PDDL3 (Gerevini & Long 2005) is an extension of previous planning languages that includes facilities for expressing preferences. It was designed in conjunction with the 2006 International Planning Competition. One of the key features of PDDL3 is that it supports temporally extended preference statements, i.e., statements that express preferences over sequences of events. In particular, in the *qualitative preferences* category of the planning competition preferences can be expressed with temporal formulae that are a subset of LTL (linear temporal logic). A plan satisfies a preference whenever the sequence of states generated by the plan's execution satisfies the LTL formula representing the preference.

PDDL3 allows each planning instance to specify a problem-specific metric used to compute the value of a plan. For any given plan, over the course of its execution various preferences will be violated or satisfied with some preference perhaps being violated multiple times. The plan value metric can depend on the preferences that are violated and the number of times that they are violated. The aim in solving the planning instance is to generate a plan that has the best metric value, and to do this the planner must be able to "monitor" the preferences to determine when and how many times different preferences are being violated. Furthermore, the planner must be able to use this information to guide its search so that it can find best-value plans.

We have crafted a preference planner that uses various techniques to find best-value plans. Our planner is based on the TLPLAN system (Bacchus & Kabanza 1998), extending TLPLAN so that fully automated heuristic-guided search for a best-value plan can be performed. We use two techniques to obtain heuristic guidance. First, we translate temporally extended preference formulae into nondeterministic finite state automata that are then encoded as a new set of predicates and action effects. When added to the existing predicates and actions, we thus obtain a new planning domain containing only standard ADL-operators. Second, once we have recovered a standard planning domain we can use a modified relaxed plan heuristic to guide search. In what follows, we describe our translation process and the heuristic search techniques we use to guide planning. We conclude with a brief discussion of related work.

Translation of LTL to Finite State Automata

TLPLAN already has the ability to evaluate LTL formulae during planning. It was originally designed to use such formulae to express search control knowledge. Thus one could simply express the temporally extended preference formulae in TLPLAN directly and have TLPLAN evaluate these formulae as it generates plans. The difficulty, however, is that this approach is by itself not able to provide any heuristic guidance. That is, there is no obvious way to use the partially evaluated LTL formulae maintained by TLPLAN to guide the planner towards satisfying these formulae (i.e., to satisfy the preferences expressed in LTL).

Instead our approach is to use the techniques presented in (Baier & McIlraith 2006) to convert the temporal formulae into nondeterministic finite state automata. Intuitively the states of the automata "monitor" progress towards satisfying the original temporal formula. In particular, as the world is updated by actions added to the plan, the state of the automata is also updated dependent on changes made to the world. If the automata enters an accepting state then the sequence of worlds traversed by the partial plan has satisfied the original temporal preference formula.

There are various issues involved in building efficient automata from an arbitrary temporal formula, and more details are provided in (Baier & McIlraith 2006). However, once the automaton is built, we can integrate it with the planning domain by creating an augmented planning domain. In the augmented domain there is a predicate specifying the current set of states that the automata could be in (it is a non-deterministic automata so there are a set of current states). Moreover, for each automata, we have a single predicate (the *accepting predicate*) that is true iff the automata has reached an accepting condition, denoting satisfaction of the preference. In addition, we define a post-action update sequence of ADL operators, which take into account the changes just made to the world and the current state of the automata in order to compute the new set of possible automata states. This post-action update is performed immediately after any action of the domain is performed. TLPLAN is then asked to generate a plan using the new augmented domain.

To deal with multiple preference statements, we apply this method to each of the preferences in turn. This generates multiple automata, and we combine all of their updates into a single ADL action (actually to simplify the translation we use a pair of ADL actions that are always executed in sequence).

A number of refinements must be made however to deal with some of the special features of PDDL3. First, in PDDL3 a preference can be scoped by a universal quantifier. Such preferences act as parameterized preference statements, representing a set of individual preference statement one for each object that is a legal binding of the universal variable. To avoid the explosion of automata that would occur if we were to generate an distinct automata for each binding, we translate such preferences into "parameterized" automata. In particular, instead of having a predicate describing the current set of states the automata could be in, we have a predicate with extra arguments which specifies what state the automata could be in for different objects. Similarly, the automata update actions generated by our translator are modified so that they can handle the update for all of the objects through universally quantified conditional effects.

Second, PDDL3 allows preference statements in action preconditions. These preferences refer to conditions that must ideally hold true immediately before performing an action. These conditions are not temporal, i.e., they refer only to the state in which the action is performed. Therefore, we do not model these preferences using automata but rather as conditional effects of the action. If the preference formula does not hold and the action is performed, then, as an effect of the action, a counter is incremented. This counter, representing the number of times the precondition preference is violated, is used to compute the metric function, described below.

Third, PDDL3 specifies its metric using an "is-violated" function. The is-violated function takes as an argument the name of a preference type, and returns the number of times preferences of this type were violated. Individual preferences are either satisfied or violated by the current plan. However, many different individual preferences can be grouped into a single type. For example, when a preference is scoped by a universal quantifier, all of the individual preference statements generated by different bindings of the quantifier yield a preference of the same type. Thus the is-violated function must be able to count the number of these preferences that are violated. Similarly, action precondition preferences can be violated multiple times, once each time the action is executed under conditions that violated the precondition preference. The automata we construct utilizes TLPLAN's ability to manipulate functions to keep track of these numbers.

Finally, PDDL3 allows specification of hard temporal constraints, which can also be viewed as being hard temporally extended goals. We also translate these constraints into automata. The accepting predicate of these automata are then treated as additional final-state goals. Moreover, we use TLPLAN's ability to incrementally check temporal constraints to prune from the search space those plans that already have violated the constraint.

Heuristic Search

The new augmented planning domain no longer has temporally extended preferences. Instead, the domain is much like a standard planning domain. Thus, we can compute relaxed plans and use those relaxed plans to compute heuristics.

In particular, we have augmented TLPLAN to allow it to compute relaxed state sequences: sequences of states that can be generated from the current state when ignoring the delete effects of actions. Notice that since the automata predicates are part of the new domain, the relaxed state sequences include predicates describing the "relaxed state" of the automata. Thus in the relaxed sequence of states not only can we compute various goal distance functions, but we can also compute various functions that depend on automata states. That is, we can compute information about the distance to satisfying various preferences. Since each preference is given a different weight in valuing a plan we can even weight the "distance to satisfying a preference" differently depending on the value of the preference.

Specifically, our heuristic function is a combination of the following functions, which are evaluated over partial plans. (We continue to work on these functions.)

Goal distance A function that is a measure of how hard it is to reach the goal. It is computed using the relaxed plan graph (similar to the one used by the FF planner (Hoffmann & Nebel 2001)). It computes a heuristic distance to the goal facts using a variant of the heuristic proposed by (Zhu & Givan 2005). The exact value of the k exponent in this heuristic is still being finalized.

Preference distance A measure of how hard it is to reach the preference goals, i.e., how hard it is to reach the accepting states of the various preference automata. Again, we use Zhu & Givan's heuristic to compute this distance.

Optimistic metric A lower bound[1] for the metric function

[1]Without loss of generality, we assume that we are minimizing the metric function.

of any plan that completes the partial plan, i.e., the best metric value that the partial plan could possibly achieve if completed to satisfy the goal. We compute this number assuming that no precondition preferences will be violated in the future, and assuming that all temporal formulae that are not currently violated by the partial plan will be true in the completed plan. To determine whether a temporal formula is not violated by the partial plan, we simply verify that its automaton is currently in a state from which there is a path to an accepting state. Finally, we assume that the goal will be satisfied at the end of the plan.

Discounted metric A weighting of the metric function evaluated in the relaxed states. Let $M(s_0)$ be the metric value of a state s_0, and $s_1, \ldots, s_n$ be the relaxed states reachable from state s until a fixed point is found. The discounted metric for s and discount factor r, $D(s, r)$, is computed as:

$$D(s,r) = M(s_0) + \sum_{i=0}^{n-1}(M(s_{i+1}) - M(s_i))r^i.$$

The factor of r we are finally going to use is not yet decided.

The final heuristic function is obtained by a combination of the functions defined above.

Our planner is able to return plans with incrementally improving metric value. It does best-first search using the heuristic described above. At all times, it keeps the metric value of the best plan found so far. Additionally, the planner prunes from the search space all those plans whose optimistic metric is worse than the best metric found so far. This is done by dynamically adding a new TLPLAN hard constraint into the planning domain.

Discussion

The technique we use to plan with temporally extended preferences presents a novel combination of techniques for planning with temporally extended goals, and for planning with preferences.

A key enabler of our planner is the translation of LTL preference formulae into automata, exploiting work described in (Baier & McIlraith 2006). There are several papers that address related issues. First is work that compiles temporally extended goals into classical planning problems such as that of Rintanen (Rintanen 2000), and Cresswell and Coddington (Cresswell & Coddington 2004). Second is work that exploits automata representations of temporally extended goals (TEGs) in order to plan with TEGs, such as Kabanza and Thiébaux's work on TLPLAN (Kabanza & Thiébaux 2005) and work by Pistore and colleagues (Lago, Pistore, & Traverso 2002). A more thorough discussion of this work can be found in (Baier & McIlraith 2006).

There is also a variety of previous work on planning with preferences. In (Bienvenu, Fritz, & McIlraith 2006) the authors develop a planner for planning with temporally extended preferences. Their planner performs best first-search based on the optimistic and pessimistic evaluation of partial plans relative to preference formulae. Preference formulae are evaluated relative to partial plans and the formulae progressed, in the spirit of TLPLAN, to determine aspects of the formulae that remain to be satisfied. Also noteworthy is the work of Son and Pontelli (Son & Pontelli 2004) who have constructed a planner for planning with temporally extended goals using answer-set programming (ASP). Their work holds promise however ASP's inability to deal efficiently with numbers has hampered their progress. Brafman and Chernyavsky (Brafman & Chernyavsky 2005) recently addressed the problem of planning with preferences by specifying qualitative preferences over possible goal states using TCP-nets. Their approach to planning is to compile the problem into an equivalent CSP problem, imposing variable instantiation constraints on the CSP solver, according to the TCP-net. This is a promising method for planning, though at the time of publication of their paper, their planner did not deal with temporal preferences.

References

Bacchus, F., and Kabanza, F. 1998. Planning for temporally extended goals. *Ann. of Math Art. Int.* 22(1-2):5–27.

Baier, J. A., and McIlraith, S. 2006. Planning with first-order temporally extended goals. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06)*. To appear.

Bienvenu, M.; Fritz, C.; and McIlraith, S. 2006. Planning with qualitative temporal preferences. In *Proceedings of the Tenth International Conference on Knowledge Representation and Reasoning (to appear)*.

Brafman, R., and Chernyavsky, Y. 2005. Planning with goal preferences and constraints. In *Proceedings of The International Conference on Automated Plann ing and Scheduling*.

Cresswell, S., and Coddington, A. 2004. Compilation of LTL goal formulas into PDDL. In *ECAI-04*, 985–986.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences for pddl3. Technical Report 2005-08-07, Department of Electronics for Automation, University of Brescia, Brescia, Italy.

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

Kabanza, F., and Thiébaux, S. 2005. Search control in planning for temporally extended goals. In *Proc. ICAPS-05*.

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

Rintanen, J. 2000. Incorporation of temporal logic control into plan operators. In *Proc. ECAI-00*, 526–530.

Son, T., and Pontelli, E. 2004. Planning with preferences using logic programming. In Lifschitz, V., and Niemela, I., eds., *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-2004)*, number 2923 in Lecture Notes in Computer Science. Springer. 247–260.

Zhu, L., and Givan, R. 2005. Simultaneous heuristic search for conjunctive subgoals. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-2005)*, 1235–1241.

Yochan$^{\mathcal{PS}}$: PDDL3 Simple Preferences as Partial Satisfaction Planning

J. Benton & Subbarao Kambhampati
Computer Sci. & Eng. Dept.
Arizona State University
Tempe, AZ 85287
{j.benton,rao}@asu.edu

Minh B. Do
Embedded Reasoning Area
Palo Alto Research Center
Palo Alto, CA 94304
minhdo@parc.com

Introduction

Yochan$^{\mathcal{PS}}$ compiles a problem using PDDL3 "simple preferences" (PDDL3-SP), as defined in the 5th International Planning Competition (IPC5), into a partial satisfaction planning (PSP) (van den Briel *et al.* 2004). The commonality of the semantics between these problem types enable the conversion. In particular, both planning problem definitions include relaxations on goals and both define plan quality metrics. We take advantage of these commonalities and produce a problem solvable by PSP planners from a PDDL3-SP problem definition. A minor restriction is made of resulting PSP plans so the compilation may be simplified to avoid extraneous exponential increases in the number of actions. We chose *Sapa*$^{\mathcal{PS}}$ to solve the new problem.

PSP Net Benefit and PDDL3-SP

In partial satisfaction planning (Smith 2004; van den Briel *et al.* 2004), goals $g \in G$ have utility values $u(g) \geq 0$, representing how much each goal is worth to a given user. Each action $a \in A$ has an associated positive execution cost c_a where A is the set of all actions in the domain. Moreover, not all goals in G need to be achieved. Let P be the lowest cost plan that achieves a subset $G' \subseteq G$ of those goals. The objective is to maximize the *net benefit*, that is tradeoff between total utility $u(G')$ of G' and total cost of actions $a \in P$:

$$maximize_{G' \subseteq G} \;\; u(G') - \sum_{a \in P} c_a \qquad (1)$$

In PDDL3 "simple preferences" (PDDL-SP), preferences can be defined in goal conditions $g \in G$ and action preconditions $pre(a) \mid a \in A$ (Gerevini & Long 2005). Conditions defined in this way do not need to be achieved for a plan to be valid. This relates well to goals as defined in PSP. However, unlike PSP, cost is acquired by failing to satisfy preferences. There is also no explicit utility defined. Let Φ be a preference condition, then $Cost(\Phi) = \alpha$, where α is a constant value.[1] Let $pref(G)$ be the set of all preference conditions on goals and $pref(a)$ be all preference preconditions on $a \in A$. For a plan P, if a preference precondition, $pref_p \in pref(a)$ where $a \in P$, is applied in state S, without satisfying p then cost $Cost(pref_p)$ is incurred. In the case of a preference on a goal, $pref_g \in pref(G)$, cost $Cost(pref_g)$ is applied when the preference goal is not satisfied at the end state of a plan. In PDDL3-SP, we want to find a plan P that incurs the least cost.

[1] In PDDL3, many preferences may have the same name. For PDDL3-SP, this is syntactic sugar and we therefore refer to preferences as if each is uniquely identified to simplify the discussion.

Compiling PDDL3-SP to PSP

Both PSP and PDDL3-SP use a notion of cost on actions, though their view differs on how to define cost. PSP defines cost directly on each action, while PDDL3-SP uses a less direct approach by defining conditions for when cost is generated. In one sense, PDDL3-SP can be viewed as considering action cost as a conditional effect on an action where cost is increased on the preference condition's negation. We use this observation to inspire our action compilation to PSP. That is, we compile PDDL3 "simple preferences" on actions in a manner that is similar to how (Gazen & Knoblock 1997) compiles conditional effects.

We handle goal preferences differently. In PSP, we gain utility for achieving goals. In PDDL3-SP, we add cost for failing to achieve goals. Taken apart these concepts are complements of one another (i.e. cost for failing and utility for succeeding). The idea is that *not* failing to achieve a goal reduces our cost (i.e. gains utility for us). Therefore, as part of our compilation to PSP we transform a "simple preference" goal to an equivalent goal with utility equal the cost produced for not satisfying it in the PDDL3-SP problem. In this way we can view goal achievement as *canceling out* the cost of obtaining the goal. That is, we can compile a goal preference $pref_p$ to an action that takes p as a condition. The effect of the action would be that we "have the preference" and hence we would place that effect in our goal state with a utility equal to $Cost(pref_p)$.

Figure 1 shows the algorithm for compiling a PDDL3-SP problem into a PSP problem. We begin by first creating a temporary action a for every preference $pref_p$ in the goals. The action a has p as a precondition, and a new effect, g_p. g_p takes the name of $pref_p$. We then add g_p to the goal set G, and give it utility equal the cost of violating the preference. The process then removes $pref_p$ from the goal set.

After processing the goals into a set of actions and new goals, we proceed by compiling each action in the problem. For each $a \in A$ we take each set $precSet$ of the power set $P(pref(a))$. This allows us to create a version

```
forall $pref(p) \in pref(G)$ do
  $pre(a) := p$
  $g_p := name(pref_p)$
  $eff(a) := g_p$
  forall $b \in A$ do
    $eff(b) := eff(b) \cup \neg\{g_{pref}\}$
  endfor;
  $A := A \cup \{a\}$
  $U(g_{pref}) := Cost(pref_p)$
  $G := (G \cup \{g_{pref}\}) \setminus \{p\}$
endfor;
$i := 0$
forall $a \in A$ do
  for each $precSet \in P(pref(a))$ do
    $pre(a_i) := pre(a) \cup precSet$
    $eff(a_i) := eff(a)$
    $c_{a_i} := Cost(pref(a) \setminus precSet)$
    $A := A \cup \{a_i\}$
    $i := i + 1$
  endfor;
  $A := A \setminus \{a\}$
endfor;
```

Figure 1: PDDL3-SP to PSP compilation process.

of a for every combination of its preferences. The cost of the action is the cost of failing to satisfy the preferences in $pref(a) \setminus precSet$. We remove a from the domain after all of its compiled actions are created. Notice that because we use the power set of preferences, this results in an exponential increase in the number of actions.

When we output a plan, we must remove all new actions that produce preference goals and our metric value is calculated as follows:

$$\sum_{g \in G} U(g) \; - \sum_{g' \in G'} U(g') \; + \sum_{a \in P} c_a \qquad (2)$$

Plan Criteria

The reader may notice that the above algorithm will generate a set of actions A_a from an original action a that are all applicable in states where all preferences are met. That is, actions that have cost may be inappropriately included in the plan at such states. This would mean that the PSP compilation could produce incorrect metric values in the final plan. One way to fix this issue would be to explicitly negate the preference conditions that are not included in the new action preconditions. This is similar to the approach taken in (Gazen & Knoblock 1997) for conditional effects. We decided against this for three related reasons. First, all known PSP planners require domains be specified using STRIPS actions and this technique would introduce non-STRIPS actions–specifically, actions with negative preconditions and those with disjunctive preconditions (due to the negation of conjunctive preferences). Second, compiling disjunctive preconditions to STRIPS may require an exponential number of new actions (Gazen & Knoblock 1997; Nebel 2000) and since we are already potentially adding an exponential number of actions in the compilation from preferences, we thought it best to avoid adding more. Lastly, and most importantly, we can use a simple criteria on the plan that removes the need to include the negation of preference conditions: We require that for every action generated from a, only the *least cost* applicable action $a_i \in A_a$ can be included in P at a given state. This criteria is already inherent in some PSP planners such as *Sapa*$^{\mathcal{PS}}$ (Do & Kambhampati 2004) and *OptiPlan* (van den Briel *et al.* 2004).

Example

As an example, let us see how an action with a preference would be compiled. Consider the following PDDL3 action taken from the IPC5 TPP domain:

```
(:action drive
 :parameters
   (?t - truck ?from ?to - place)
 :precondition (and
   (at ?t ?from) (connected ?from ?to)
   (preference p-drive (and
    (ready-to-load goods1 ?from level0)
    (ready-to-load goods2 ?from level0)
    (ready-to-load goods3 ?from level0))
    ))
 :effect (and (not (at ?t ?from))
              (at ?t ?to)))
```

A plan metric assigns a weight to our preferences:

```
(:metric (+ (* 10 (is-violated p-drive) )
            (* 5  (is-violated P0A) )))
```

This action can be compiled into PSP style actions:

```
(:action drive-0
 :parameters
   (?t - truck ?from ?to - place)
 :precondition (and
   (at ?t ?from) (connected ?from ?to)
    (ready-to-load goods1 ?from level0)
    (ready-to-load goods2 ?from level0)
    (ready-to-load goods3 ?from level0)))
 :effect (and (not (at ?t ?from))
              (at ?t ?to)))

(:action drive-1
 :parameters
   (?t - truck ?from ?to - place)
 :cost 10
 :precondition (and
   (at ?t ?from) (connected ?from ?to))
 :effect (and (not (at ?t ?from))
              (at ?t ?to)))
```

Let us also consider the following goal preference in the same domain:

```
(:goal
(preference P0A (stored goods1 level1)))
```

The goal will be compiled into the following PSP action:

```
(:action p0a
 :parameters ()
 :precondition (and (stored goods1 level1))
 :effect (and (hasPref-p0a) ) )
```

With the goal:

```
((hasPref-p0a) 5.0)
```

5th International Planning Competition

For the planning competition, we used the compilation described in combination with *Sapa*$^{\mathcal{PS}}$ (Do & Kambhampati 2004) to create *Yochan*$^{\mathcal{PS}}$. *Sapa*$^{\mathcal{PS}}$ inherently meets the plan criteria required for our compilation. It performs an A* search, and its cost propagated relaxed planning graph heuristic ensures that, given any set of actions with the same effects, the branch with the least cost action will be taken. As another point, *Sapa*$^{\mathcal{PS}}$ is capable of handling "hard" goals, which are prevalent in the competition domains. It has also shown to be successful in solving PSP problems (van den Briel *et al.* 2004).

Conclusion

We outlined a method of converting domains specified in the "simple preferences" category of the Fifth International Planning Competitions (PDDL3-SP) to partial satisfaction planning (PSP) problems. The technique uses ideas for compiling action conditional effects into STRIPS actions as a basis. Though the process has the potential for adding several actions to the domain, in practice the number of added actions appears manageable.

References

Do, M., and Kambhampati, S. 2004. Partial satisfaction (over-subscription) planning as heuristic search. In *Knowledge Based Computer Systems*.

Gazen, B., and Knoblock, C. 1997. Combining the expressiveness of ucpop with the efficiency of graphplan. In *Fourth European Conference on Planning*.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3: The language of the fifth international planning competition. Technical report, University of Brescia, Italy.

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

Smith, D. 2004. Choosing objectives in over-subscription planning. In *Proc. of ICAPS-04*.

van den Briel, M.; Sanchez, R.; Do, M. B.; and Kambhampati, S. 2004. Effective approaches for partial satisfaction (over-subscription) planning. In *Proc. of AAAI-04*.

IPPLAN: Planning as Integer Programming

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

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

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

Overview

IPPLAN is an integer programming based planning system. It builds on the previous work of planning as integer programming, including that of: ILP-PLAN by Kautz and Walser (1999), the state change encoding by Vossen *et al.* (1999), Optiplan by van den Briel and Kambhampati (2005), and most significantly the state change flow encodings by van den Briel, Vossen, and Kambhampati (2005). Moreover, it adds on to the existing planning compilation approaches, including that of: SATPLAN by Kautz and Walser (1992), and GP-CSP by Do and Kambhampati (2000).

The current version of IPPLAN consists of two separate modules: (1) a translator written in Python, and (2) an integer programming modeler written in C++.

In order to solve a planning problem, the two modules are run consecutively. The translator is run first, and transforms a PDDL input into a state variable representation based on the SAS+ formalism. The integer programming modeler is run second, and generates the needed data structures and formulates the planning problem as an integer programming problem. The resulting integer programming problem is then solved using CPLEX (ILOG 2002).

The translator is an extension to the preprocessing algorithm of MIPS (Edelkamp & Helmert 1999). It was designed and developed by Helmert (2006) as one of the components for the Fast Downward planner. The translator is a stand alone component and therefore can easily be incorporated into other applications. The purpose of the translator is to ground all operators and axioms, convert the propositional (binary) representation to a state variable (multi-valued) representation of the planning problem, and to compile away most of the ADL features. A detailed description of the translator and its translation algorithm is described by Helmert (2006).

IPPLAN can support a collection of integer programming formulations. Currently, IPPLAN supports the One State Change (1SC) and the Generalized One State Change (G1SC) formulations as described by van den Briel, Vossen, and Kambhampati (2005). Both these formulations are restricted to solve propositional planning problems only, so currently IPPLAN is a propositional planning system. In the future, however, we would like to add more formulations to IPPLAN and broaden the scope of planning problems that it can handle.

When the 1SC formulation is used IPPLAN will find optimal makespan plans. With the G1SC formulation IPPLAN will not guarantee optimality, but generally find plans with few number of actions. In both these formulations state changes in the state variables are modeled as flows in an appropriately defined network. As a consequence, the integer programming formulations can be interpreted as a network flow problems with additional side constraints.

IPPLAN uses CPLEX (ILOG 2002) for solving the integer programming problems. CPLEX is a commercial software package that solves linear programming, mixed integer programming, network flow, and convex quadratic programming problems.

References

Do, M., and Kambhampati, S. 2000. Solving planning graph by compiling it into a CSP. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000)*, 82–91.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proceedings of the European Conference on Planning (ECP-99)*, 135–147. Springer-Verlag.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 25:(Accepted for publication).

ILOG Inc., Mountain View, CA. 2002. *ILOG CPLEX 8.0 user's manual.*

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *Proceedings of the European Conference on Artificial Intelligence (ECAI-1992).*

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

van den Briel, M., and Kambhampati, S. 2005. Optiplan: Unifying IP-based and graph-based planning. *Journal of Artificial Intelligence Research* 24:623–635.

van den Briel, M.; Vossen, T.; and Kambhampati, S. 2005. Reviving integer programming approaches for ai planning: A branch-and-cut framework. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-2005)*, 161–170.

Vossen, T.; Ball, M.; Lotem, A.; and Nau, D. 1999. On the use of integer programming models in AI planning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 304–309.

Large-Scale Optimal PDDL3 Planning with MIPS-XXL

Stefan Edelkamp[1], Shahid Jabbar[2], and Mohammed Nazih[3] *
Computer Science Department
University of Dortmund, Dortmund, Germany

Introduction

State trajectory and preference constraints are the two language features introduced in PDDL3 (Gerevini & Long 2005) for describing benchmarks of the 5th international planning competition. *State trajectory constraints* provide an important step of the agreed fragment of PDDL towards the description of *temporal control knowledge* and *temporally extended goals* They assert conditions that must be met during the execution of a plan and are often expressed using quantification over domain objects.

We suggest to compile the state trajectory and preference constraints into PDDL2 (Edelkamp 2006). Trajectory constraints are compiled into Büchi automata that are synchronized with the exploration of the planning problem, while preference constraints are transformed into numerical fluents that are changed upon violation. An internal weighted best-first search is invoked that tries to find a solution. Once a solution is found, the solution quality is inserted in the problem description and a new search is started using earlier solution cost as the minimization parameter. If the internal search fails to terminate with in a specified amount of time, we switch to a cost-optimal external breadth-first search procedure that utilizes harddisk to store the generated states.

Compilation of State Trajectory Constraints

State trajectory constraints impose restrictions on plans. Their semantics can best be captured by using a special kind of automata structure called as Büchi automata. Büchi automata has long been used in automata-based model checking (Clarke, Grumberg, & Peled 2000), where both the model to be analyzed and the specification to be checked are modeled as non-deterministic *Büchi automata.* Syntactically, Büchi automata are ordinary automata, but with a special acceptance condition. Let ρ be a run and $\mathit{inf}(\rho)$ be the set of states reached infinitely often in ρ, then a Büchi automaton accepts, if the intersection between $\mathit{inf}(\rho)$ and the set of final states F is not empty. In automata-based model-checking, a specification property is falsified if and only if there is a non-empty intersection between the language accepted by the Büchi automata of the model and of the *negated* specification.

For trajectory constraints, we need a Büchi automaton for the model and one for each trajectory constraints, together with some algorithm that validates if the language intersection is not empty. By the semantics of (Gerevini & Long 2005) it is clear that all sequences are finite, so that we can interpret a Büchi automaton as a non-deterministic finite state automaton (NFA), which accepts a word if it *terminates* in a final state. The labels of such an automaton are conditions over the propositions and fluents in a given state. During the exploration, we simulate a synchronization of all Büchi automata.

To encode the simulation of the synchronized automata, we devise a predicate `(at ?n - state ?a - automata)` to be instantiated for each automata state and each automata that has been devised. For detecting accepting states, we include instantiations of predicate `(accepting ?a - automata)`.

As we require a tight synchronization between the constraint automaton transitions and the operators in the original planning space, we include *synchronization flags* that are flipped when an ordinary or a constraint automaton transition is chosen.

Compilation of Preferences

For preference p we include numerical fluents `is-violated-p` to the grounded domain description. For each operator and each preference we apply the following reasoning. If the preferred predicate is contained in the *delete list* then the fluent is increased, if it is contained in the *add list*, then the fluent is decreased, otherwise it remains unchanged[1].

*All three authors are supported by the German Research Foundation (DFG) projects *Heuristic Search* Ed 74/3 and *Directed Model Checking* Ed 74/2.

[1]An alternative semantic to (Gerevini & Long 2005) would be to set the fluent to either 0 or 1. For rather complex propositional or numerical goal conditions in a preference condition, we can use *conditional effects.*

For preferences p on a state trajectory constraint that has been compiled to an automaton a, the fluents `(is-violated-a-p)` substitute the atoms `(is-accepting-a)` in an obvious way. If the automata accepts, the preference is fulfilled, so the value of `(is-violated-a-p)` is set to 0. In the transition that newly reaches an accepting state `(is-violated-a-p)` is set to 0, if it enters a non-accepting state it is set to 1. The `skip` operator also induces a cost of 1 and the automaton moves to a dead state.

External Exploration

For complex planning problems, the size of the state space can easily surpass the main memory limits. Most modern operating systems provides a facility to use larger address spaces through *virtual memory* that can be larger than internal memory. For the programs that do not exhibit any *locality of reference* for memory accesses, such general purpose virtual memory management can instead lower down their performances.

Algorithms that explicitly manage the memory hierarchy can lead to substantial speedups, since they are more informed to predict and adjust future memory access. In (Korf & Schultze 2005) we see a complete exploration of the state space of 15-puzzle made possible utilizing a 1.4 Terabytes of secondary storage. In (Jabbar & Edelkamp 2005) a successful application of external memory heuristic search for LTL model checking is presented.

The standard model (Aggarwal & Vitter 1988) for comparing the performance of external algorithms consists of a single processor, a small internal memory that can hold up to M data items, and an unlimited secondary memory. The size of the input problem (in terms of the number of records) is abbreviated by N. Moreover, the *block size* B governs the bandwidth of memory transfers. External-memory algorithms are evaluated in terms of number of I/Os, where each block transfer amounts to one I/O.

It is convenient to express the complexity of external-memory algorithms using a number of frequently occurring primitive operations: *Scanning*, *scan(N)* with an I/O complexity of $\Theta(\frac{N}{B})$ that can be achieved through trivial sequential access; *Sorting*, *sort(N)* with an I/O complexity of $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ that can be achieved through external *Merge* or *Distribution Sort*.

Cost-Optimal External BFS

An implicit variant of Munagala and Ranade's algorithm (Munagala & Ranade 1999) for explicit BFS-search in implicit graphs has been coined to the term *delayed duplicate detection* for *frontier search*. It assumes an undirected search graph. Let $\mathcal{I}$ be the initial state, and N be the implicit successor generation function. Figure 1 displays the pseudo-code for external BFS exploration incrementally improving an upper bound U on the solution quality. The state sets corresponding to each layer are represented in form of files.

```
**Procedure Cost-Optimal-External-BFS**
  $U \leftarrow \infty$; $i \leftarrow 1$
  $Open(-1) \leftarrow \emptyset$; $Open(0) \leftarrow \{\mathcal{I}\}$
  **while** $(Open(i-1) \neq \emptyset)$
    $A(i) \leftarrow N(Open(i-1))$
    **forall** $v \in A(i)$
      **if** $v \in \mathcal{G}$ **and** $Metric(v) < U$
        $U \leftarrow Metric(v)$
        $ConstructSolution(v)$
    $A'(i) \leftarrow$ *remove duplicates from* $A(i)$
    **for** $l \leftarrow 1$ **to** $loc$
      $A'(i) \leftarrow A'(i) \setminus Open(i-l)$
    $Open(i) \leftarrow A'(i)$
    $i \leftarrow i+1$
```

Figure 1: Cost-Optimal External BFS Planning Algorithm.

The search frontier denoting the current BFS layer is tested for an intersection with the goal, and this intersection is further reduced according to the already established bound.

Layer $Open(i-1)$ is scanned and the set of successors are put into a buffer of size close to the main memory capacity. If the buffer becomes full, internal sorting followed by a duplicate elimination scanning phase generates a sorted duplicate-free state sequence in the buffer that is flushed to disk. A sets in the pseudo-code corresponds to temporary sets.

In the next step, *external merging* is applied to merge the flushed buffers into $Open(i)$ by a simultaneous scan. The size of the output files is chosen such that a single pass suffices. Duplicates are eliminated while merging. Since the files were sorted, the complexity is given by the scanning time of all files. One also has to eliminate the previous layers from $Open(i)$ to avoid recomputations. The number of previous layers that have to be subtracted are dependent on the *locality*(loc) of the graph. In case of undirected graphs, two layers are sufficient. For directed graphs, we suggest to calculate this parameter by searching for a sequence of operators that when applied to a state produces no effect. Such a sequence can be computed by just looking at all possible sequences of operators. The length of the shortest such sequence dictates the locality of a planning graph. The process is repeated until $Open(i-1)$ becomes empty, or the goal has been found.

The I/O Complexity of External BFS for undirected graph can be computed as follows. The successor generation and merging involves $O(sort(|N(Open(i-1))|) + (\sum_{l=1}^{loc} scan(|Open(i-l)|)$ I/Os. However, since $\sum_i |N(Open(i))| = O(|E|)$ and $\sum_i |Open(i)| = O(|V|)$, the total execution time is $O(sort(|E|) + loc \cdot scan(|V|))$ I/Os.

In an internal non memory-limited setting, a plan is constructed by backtracking from the goal node to the start node. This is facilitated by saving with every node a pointer to its predecessor. However, there is one subtle problem: predecessor pointers are not available on disk. This is resolved as follows. Plans are reconstructed by saving the predecessor together with every state, by using backtracking along the stored files, and by looking for matching predecessors. This results in a I/O complexity that is at most linear to the number of stored states.

In planning with preferences, we often have a monotone decreasing instead of a monotonic increasing cost function. Hence, we cannot prune states with an evaluation larger than the current one. Essentially, we are forced to look at all states. In order to speed up the external search with a compromise on the optimality, we can apply a procedure similar to beam-search where we can limit our search to expand only a small portion of the best nodes within each layer. On competition problems, we have managed to have good accelerations through this approach.

Implementation

We first transform PDDL3 files with preferences and state trajectory constraints to grounded PDDL3 files without them. For each state trajectory constraint, we parse its specification, flatten the quantifiers and write the corresponding LTL-formula to disk.

Then, we derive a Büchi-automaton for each LTL formula and generates the corresponding PDDL code to modify the grounded domain description[2]. Next, we merge the PDDL descriptions corresponding to Bc̈hi automata and the problem file. Given the grounded PDDL2 outcome, we apply efficient heuristic search forward chaining planner *Metric-FF* (Hoffmann 2003). Note that by translating plan preferences, otherwise propositional problems are compiled into metric ones. For temporal domains, we extended the *Metric-FF* planner to handle temporal operators and timed initial literals. The resulting planner is slightly different from known state-of-the-art systems of adequate expressiveness, as it can deal with disjunctive action time windows and uses an internal linear-time approximate scheduler to derive parallel (partial or complete) plans. The planner is capable of compiling and producing plans for all competition benchmark domains.

Due to the numerical fluents introduced for preferences, we are faced with a search space where cost is not necessarily monotone. For such state spaces, we have to look at all the states to reach to an optimal solution. The issue then arises is if it is possible to reach an optimal solution fast. We propose to use a branch-and-bound like procedure on top of the best-first weighted heuristic search as offered by the extended *Metric-FF* planning system. Upon reaching a goal, we terminate our search and create a new problem file where the goal condition is extended to minimize the found solution cost. The search is restarted on this new problem description. The procedure terminates when the whole state space is looked at. The rationale behind this is to have improved guidance towards a better solution quality. If internal search failed to terminate within a specified amount of time, we switch to external BFS search.

[2] `www.liafa.jussieu.fr/~oddoux/ltl2ba`. Similar tools include *LTL→NBA* and the never-claim converter inherent to the SPIN model checker.

Conclusions

We propose to translate temporal and preference constraints into PDDL2. Temporal constraints are converted into Büchi automata in PDDL format, and are executed synchronously with the main exploration. Preferences are compiled away by a transformation into numerical fluents that impose a penalty upon violation. Incorporating better heuristic guidance, especially, for preferences is still an open research frontier.

Search is performed in two stages. Initially, an internal best-first is invoked that keeps on improving its solution quality till the search space is exhausted. After a given time limit, the internal search is terminated and an external breadth-first search is started.

The crucial problem in external memory algorithms is the duplicate detection with respect to previous layers to guarantee termination. Using the locality of the graph calculated directly from the operators themselves, we provide a bound on the number of previous layers that have to be looked at.

Since states are kept on disk, external algorithms have a large potential for parallelization. We noticed that most of the execution time is consumed while calculating heuristic estimates. Distributing a layer on multiple processors can distribute the internal load without having any effect on the I/O complexity.

References

Aggarwal, A., and Vitter, J. S. 1988. The input/output complexity of sorting and related problems. *Journal of the ACM* 31(9):1116–1127.

Clarke, E.; Grumberg, O.; and Peled, D. 2000. *Model Checking.* MIT Press.

Edelkamp, S. 2006. On the compilation of plan constraints and preferences. In *ICAPS.* To Appear.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences for PDDL3. Technical Report R.T. 2005-08-07, Department of Electronics for Automation, University of Brescia, Brescia, Italy.

Hoffmann, J. 2003. The Metric FF planning system: Translating "Ignoring the delete list" to numerical state variables. *JAIR* 20:291–341.

Jabbar, S., and Edelkamp, S. 2005. I/O efficient directed model checking. In *Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 313–329.

Korf, R. E., and Schultze, P. 2005. Large-scale parallel breadth-first search. In *AAAI*, 1380–1385.

Munagala, K., and Ranade, A. 1999. I/O-complexity of graph algorithms. In *SODA*, 687 – 694.

Optimal Symbolic PDDL3 Planning with MIPS-BDD

Stefan Edelkamp [*]
Computer Science Department
University of Dortmund, Dortmund, Germany

Introduction

State trajectory and plan preference constraints are the two language features introduced in PDDL3 (Gerevini & Long 2005) for describing benchmarks of the 5^{th} international planning competition. *State trajectory constraints* provide an important step of the agreed fragment of PDDL towards the description of temporal control knowledge (Bacchus & Kabanza 2000) and temporally extended goals (DeGiacomo & Vardi 1999). They assert conditions that must be met during the execution of a plan and are often expressed using using quantification over domain objects. Annotating goal conditions and state trajectory constraints with *preferences* models *soft constraints*. For planning with preferences, the objective function scales the violation of the constraints.

Symbolic exploration based on BDDs (Bryant 1985) acts on sets of states rather than on singular ones and exploit redundancies in the joint state representation. BDDs are directed acyclic automata for the bitvector representation of a state. The unique representation of a state set as a BDD is much more memory-efficient than an explicit representation for the state set. In MIPS-BDD we make optimal BDD solver technology applicable to planning with PDDL3 domains. We compile state trajectory expressions to PDDL2 (Fox & Long 2003). The grounded representation is annotated with propositions that maintain the truth of preferences and operators that model that the synchronized execution or an associated property automaton. We contribute *Cost-Optimal Breadth-First-Search* and adapt it to the search with preference constraints.

Symbolic Breadth-First Search

Symbolic search is based on satisfiability checking. The idea is to make use of Boolean functions to avoid (or at least lessen) the costs associated with the exponential memory blow-up for the state set involved as problem sizes get bigger. For propositional action planning problems we can encode the atoms that are valid in a given planning state individually by using the binary representation of their ordinal numbers, or via the bit vector of atoms being true and false.

There are many different possibilities to come up with an encoding of states for a problem. The more obvious ones seem to waste a lot of space, which often leads to bad performance of BDD algorithms. We implemented the approach of (Helmert 2004) to infer a minimized finite domain encoding of a propositional planning domain[1]

Given a fixed-length binary encoding for the state vector of a search problem, characteristic functions represent state sets. The function evaluates to true for the binary representation of a given state vector, if and only if, the state is a member of that set. As the mapping is 1-to-1, the characteristic function can be identified with the state set itself. Transitions are formalized as relations, i.e., as sets of tuples of predecessor and successor states, or, alternatively, as the characteristic function of such sets. The *transition relation* has twice as many variables as the encoding of the state. If x is the binary encoding of a state and x' is the binary encoding of a successor state, then $T(x, x')$ evaluates to true. We observe that T is the disjunct of all individual state transitions T_O, with O being an operator in $\mathcal{O}$. What we are really interested in, is to compute the (partitioned) *image* $\bigvee_{O\in\mathcal{O}} \exists x\ (T_O(x,x') \wedge \mathit{Open}(x))$ of a state set represented by *Open* wrt. a transition relation T.

For symbolic breadth-first search, let Open_i be the boolean representation of a set of states reachable from the initial state $\mathcal{I}$ in i steps, initialized with $\mathit{Open}_0 = \mathcal{I}$, and $\mathit{Open}_{i+1}(x') = \bigvee_{O\in\mathcal{O}} \exists x\ (T_O(x,x') \wedge \mathit{Open}_i(x))$. Note that S on the right hand side of the equation depends on x compared to x' on the left hand side. Thus, it is necessary to substitute x' with x in Open_i, written as $\mathit{Open}_i[x \leftrightarrow x']$. To terminate the exploration, we check, whether $\mathit{Open}_i \wedge \mathcal{G}$ is equal to the *false* function $\bot$.

In order to retrieve the solution path we assume that all sets $\mathit{Open}_0, \ldots, \mathit{Open}_i$ are available. We start with a state that is in the intersection of Open_i and the goal $\mathcal{G}$. This state is the last one on the sequential optimal solution path. We take its characteristic function S into the relational product with T to compute its potential predecessors. Next we compute the second last state on the optimal solution path in the intersection of *Pred* and Open_{i-1}, and iterate until the entire solution has been constructed.

[*]The author is supported by the German Research Foundation (DFG) project *Heuristic Search* Ed 74/3

[1]We found an application for further improvement of the encoding through a specialized BDD exploration. A set of atoms $a_1 \vee \ldots \vee a_n$ can be merged to a fact/SAS$^+$ group if the planning goal $\sum_{1\leq i\neq j\leq n} a_i \wedge a_j$ cannot be reached from the initial state. A BDD backward search terminates usually fast.

We employ BDDs for symbolic exploration. A BDD is a data structure for a concise and unique representation of Boolean functions in form of a DAG with a single root node and two sinks, labeled “1” and “0”, respectively. For evaluating the represented function for a given input, a path is traced from the root node to one of the sinks. The variable ordering has a large influence on the size of a reduced and ordered BDD. In an interleaved representation, that we employ for the transition relation, we alternate between x and x' variables. Moreover, we have experimented that preference variables are better to be queried at the top of the BDD.

BDDs for Bounded Arithmetic Constraints

The computation of a BDD $F(x)$ for a linear objective function $f(x) = \sum_{i=1}^{n} a_i x_i$, we first compute the minimal and maximal value that f can take. This defines the range that has to be encoded in binary. For the ease of presentation we assume that we consider $x_i \in \{0, 1\}$.

The work of (Bartzis & Bultan 2006) shows that the BDD for representing f has at most $O(n \sum_{i=1}^{n} a_i)$ nodes and can be constructed with matching time performance. Even wile taking the most basic representation, this result improves on alternative, more expressive structures like ADDs. Moreover, the result generalizes to variables $x_i \in \{0, \dots, 2^b\}$ and the conjunction/disjunction of several linear arithmetic formulas. This implies that Metric Planning for bounded linear arithmetic expressions in the preconditions and effects is actually efficient for BDDs.

The BDD construction algorithm in MIPS-BDD for the objective function differs from the specialized construction in (Bartzis & Bultan 2006) but computes the same result.

Symbolic Cost-Optimal Breadth-First Search

We build the binary representation for the objective function as follows. For goal preferences of type (`preference` $p\ \phi_p$) we associate a Boolean variable v_p (denoting the violation of p) and construct the following indicator function: $X_p(v, x) = (v_p \wedge \phi_p(x)) \vee (\neg v_p \wedge \phi_p(x))$.

Figure 1 displays the pseudo-code for a symbolic BFS-exploration incrementally improving an upper bound U on the solution length. The state sets that are used are represented in form of BDDs. The search frontier denoting the current BFS layer is tested for an intersection with the goal, and this intersection is further reduced according to the already established bound.

Theorem The latest plan stored by the algorithm *Cost-Optimal-Symbolic-BFS* has minimal cost.

Proof The algorithm eliminates duplicates and traverses the entire planning state space. It generates each possible planning state exactly once. Only inferior states are pruned.

State Trajectory Constraints

State trajectory constraints can be interpreted Linear Temporal Logic (LTL) (Gerevini & Long 2005) and translated into automata that run concurrent to the search and accept when the constraint is satisfied (Gastin & Oddoux 2001). LTL includes temporal modalities like **A** for *always*, **F** for

```
Procedure Cost-Optimal-Symbolic-BFS
Input: State space problem with transition relation $T$
       Goal BDD $\mathcal{G}$, and initial BDD $\mathcal{I}$
Output: Optimal solution path is stored

  $U \leftarrow \infty$
  loop
    $Reach(x') \leftarrow \mathcal{I}(x')$; $Open(x') \leftarrow \mathcal{I}(x)$
    $Intersection(x) \leftarrow \mathcal{I}(x) \wedge \mathcal{G}(x)$
    $Bound(v) \leftarrow F(v) \wedge \bigvee_{i=0}^{U} [v = i]$
    $Eval(v, x) \leftarrow Intersection(x) \wedge \bigwedge_p X_p(v, x)$
    $Metric(x) \leftarrow \exists v : Eval(v, x) \wedge Bound(v)$
    while ($Metric(x) \neq \bot$)
      if ($Open = \bot$) return "Exploration completed"
      $Succ(x') = \bigvee_{O \in \mathcal{O}} \exists x\, T_O(x, x') \wedge Open(x)$
      $Open(x) \leftarrow (Succ(x') \wedge \neg Reach(x'))[x' \leftrightarrow x]$
      $Reach(x') \leftarrow Reach(x') \vee Succ(x')$
      $Intersection(x) \leftarrow Open(x) \wedge \mathcal{G}(x)$
      $Eval(v, x) \leftarrow Intersection(x) \wedge \bigwedge_p X_p(v, x)$
      $Metric(x) \leftarrow \exists v : Eval(v, x) \wedge Bound(v)$
    $U \leftarrow ConstructAndStoreSolution(Metric(x)) - 1$
```

Figure 1: Cost-Optimal BFS Planning Algorithm.

eventually, and **U** for *until*. We propose to compile the automata back to PDDL with each transition introducing a new operator (Edelkamp 2006). Each automaton state for each automaton results in an atom. For detecting accepting states we additionally include *accepting* propositions. The initial state of the planning problem includes the start state of the automaton and an additional proposition if it is accepting. For all automata, the goal includes their acceptance.

Including state trajectory constraints in the Cost-Optimal Breadth-First Search algorithm is achieved as follows.

For (`hold-after` $t\ \phi$) we impose that ϕ is satisfied for the search frontier in all steps $i > t$. For (`hold-during` $t_1\ t_2\ \phi$) as similar reasoning applies.

For (`sometimes` ϕ) we apply automata-based model checking to build a (Büchi) automata for the LTL formula $\mathbf{F}\phi$. Let $\mathcal{S}$ be the original planning space and $A_{\mathbf{F}\phi}$ be the constructed (Büchi) automaton for formula $A_{\mathbf{F}\phi}$ and $\otimes$ the cross product between two automata, then $\mathcal{P} \leftarrow \mathcal{P} \otimes A_{\mathbf{F}\phi}$ and $\mathcal{G} \leftarrow \mathcal{G} \cup \{\texttt{accepting}(A_\phi)\}$. The initial state is extended by the initial state of the automaton, which in this case is not accepting.

For (`sometimes-before` $\phi\ \psi$) the temporal formula is more complicated, but the reasoning remains the same. We compile $\mathcal{P} \leftarrow \mathcal{P} \otimes A_{(\neg\phi\wedge\neg\psi)\mathbf{U}((\neg\phi\wedge\psi)\vee(\mathbf{A}(\neg\phi\wedge\neg\psi)))}$ and adapt the planning goal and the initial state accordingly.

For (`always` ϕ) we apply automata theory to construct $\mathcal{P} \leftarrow \mathcal{P} \otimes A_{\mathbf{G}\phi}$. Alternatively, for all i we could impose $Open_i \leftarrow Open_i \wedge \phi$ in analogy to `hold-during` and `hold-after`. For (`at-most-once` ϕ) we assign the planning problem $\mathcal{P}$ to $\mathcal{P} \otimes A_{\mathbf{A}\phi \rightarrow (\phi\mathbf{U}(\mathbf{G}\neg\phi)))}$. For (`within` $t\ \phi$) we build the cross product $\mathcal{P} \leftarrow \mathcal{P} \otimes A_{\mathbf{F}\phi}$. Moreover, we set $Open_t \leftarrow Open_t \wedge \{\texttt{accepting}(A_{\mathbf{F}\phi})\}$.

Preferences for State Trajectory Constraints

For state trajectory constraints that are constructed via automata theory, we apply the following construction. Instead of adding the automaton acceptance to the goal state we combine the acceptance with the violation predicate. If the automaton accepts then the preference is not violated; if it is located in a non-accepting state, then it is violated. For example, given (`preference` p (`at-most-once` ϕ)) we explore the cross product $\mathcal{P} \leftarrow \mathcal{P} \otimes A_{\mathbf{A}\phi\rightarrow(\phi\mathbf{U}(\mathbf{G}\neg\phi))}$. Let $a = \{\texttt{accepting}(A_{\mathbf{A}\phi\rightarrow(\phi\mathbf{U}(\mathbf{G}\neg\phi))})\}$. If $a \in \mathit{add}(O)$ then $\mathit{del}(O) \leftarrow \mathit{del}(O) \cup \{v_p\}, \mathit{add}(O) \leftarrow \mathit{add}(O) \setminus \{v_p\}$. If $a \in \mathit{del}(O)$ then $\mathit{add}(O) \leftarrow \mathit{add}(O) \cup \{v_p\}, \mathit{del}(O) \leftarrow \mathit{del}(O) \setminus \{v_p\}$. An specialized operator *skip* allows to fail the automata completely. If automaton is ignored once, it remains invalid for the rest of the computation.

Memory Limitation

BDDs already save space for large state sets. For purely propositional domains we additionally apply bidirectional symbolic BFS, which is often much faster as unidirectional search. Symbolic BFS is supposed to have small search frontiers (Jensen *et al.* 2006).

One implemented idea is an extension to *Frontier-Search* (Korf *et al.* 2005), which has been proposed for undirected or directed acyclic graph structures. In more general planning problems we have established that a duplicate detection scope (a.k.a. *locality*) of 4 is sufficient to guarantee termination for *Cost-Optimal-Symbolic-BFS* in the competition domains. Moreover, we do not store any intermediate BDD layer that corresponds to state trajectory automata transitions. Only the layers that correspond to the original unconstrained state space are stored.

Our competition results are either *step-optimal* (*Propositional* domains) or *cost-optimal* (*Simple Preferences* / *Qualitative Preferences* domains). We have not yet implemented support for metric and temporal planning operators. There is 3 restrictions to the optimality in state-trajectory domains.

1. We do not support *preference preconditions*. Actually, we can parse and process the conditions, but as the domain of the `is-violated` variables is in fact unbounded this affects a possible encoding as a BDD. Nonetheless, as these variables are monotone increasing, it is not difficult to design a specialized solution for them.
2. We assume that the automaton that is built does not affect the optimality. An automaton that constructed via the LTL translation in LTL2BA is in fact optimized in the number of states and not for the preserving path lengths. On the other hand, there some LTL converters that preserve optimal paths (Schuppan & Biere 2005).
3. The exploration is terminated by limited time or space resources. In this case the reported plans for preference domains are optimal only wrt. the search depth reached.

For larger problems, we looked at suboptimal solutions. We have tested an in-built support for canceling the exploration if the BDD node count for optimal search exceeds a threshold on BDD nodes that corresponds to the limitations of main memory. Subsequently, the entire memory for all BDD nodes is released. We successfully tested two strategies, *heuristic symbolic search* based on pattern databases and *symbolic beam-search* removing unpromising states. For the competition, we switched this feature off.

Conclusion

We have devised an optimal propositional PDDL3 planning algorithm based on BDDs. Besides using the same LTL2BA converter, the algorithm shares no code with our explicit-state planner MIPS-XXL. As the approach for state trajectory constraints relies on a translation to LTL, it has the potential to deal with much larger temporal constraint language expressiveness than currently under consideration.

After the competition, we will likely extend the above planning approach to general domains with linear expressions in the actions. As a prerequisite to apply (Bartzis & Bultan 2006) numerical state variables have to fit into some finite domains. Most of the metric planning domains around belong to this group. Moreover, we encountered that model checkers like nuSMV and CadenceSMV can already deal with LTL formula. For this cases, the LTL formula is directly encoded into a transition relation without using an intermediate explicit automaton (Schuppan & Biere 2005).

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116:123–191.

Bartzis, C., and Bultan, T. 2006. Efficient BDDs for bounded arithmetic constraints. *STTT* 8(1):26–36.

Bryant, R. E. 1985. Symbolic manipulation of boolean functions using a graphical representation. In *ACM/IEEE DAC*, 688–694.

DeGiacomo, G., and Vardi, M. Y. 1999. Automata-theoretic approach to planning for temporally extended goals. In *ECP*, 226–238.

Edelkamp, S. 2006. On the compilation of plan constraints and preferences. In *ICAPS*, To Appear.

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.

Gastin, P., and Oddoux, D. 2001. Fast LTL to Büchi automata translation. In *CAV*, 53–65.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3. Technical report, Department of Electronics for Automation, University of Brescia.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *ICAPS*, 161–170.

Jensen, R.; Hansen, E.; Richards, S.; and Zhou, R. 2006. Memory-efficient symbolic heuristic search. In *ICAPS*, To Appear.

Korf, R. E.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the ACM* 52(5):715–748.

Schuppan, V., and Biere, A. 2005. Shortest counterexamples for symbolic model checking of LTL with past. In *TACAS*, 493–509.

FDP: Filtering and Decomposition for Planning

Stéphane Grandcolas et Cyril Pain-Barre
LSIS – UMR CNRS 6168
Domaine Universitaire de Saint-Jérôme
Avenue Escadrille Normandie-Niemen
13397 MARSEILLE CEDEX 20 France
{stephane.grandcolas,cyril.pain-barre}@lsis.org

Overview

FDP is a planning system based on the paradigm of planning as constraint satisfaction, that searches for optimal sequential plans. The input langage is PDDL with typing and equality. FDP works directly on a structure related to Graphplan's planning graph: given a fixed bound on the length of the plan, the graph is incrementally built. Each time the graph is extended, a search for a sequential plan is made.

FDP does not use any external solver. The reason is that using an up-to-date CSP solver allows to take benefits from recent advances in the CSP field, but has also the disadvantage that the resulting system can not take into account the specificities of planning nor the structure of the problem. Hence, as the DPPLAN system (Baioletti, Marcugini, & Milani 2000), FDP integrates consistency rules and filtering and decomposition mechanisms suitable for planning.

A structure that represents the planning problem is incrementally extended until a solution is found or a fixed bound of the number of steps is reached. The current implementation extends the structure with one step more. Each time a depth-first search is performed, based on problem decomposition with actions sets partitioning. Nevertheless, it is basically *Depth-First Iterative Deepening* (Korf 1985) (or IDA^* with admissible heuristic of constant cost 1).

FDP does not detect unsolvability of problems, as many other similar approaces (Rintanen 1998; Baioletti, Marcugini, & Milani 2000; Lopez & Bacchus 2003). Then, it must be given a fixed bound of plan length in order to stop on unsolvable instances of problems. This weakness of the algorithm will be adressed in future work.

The search procedure is complete. Then if a solution is found, it is minimal in terms of plan length. On the other hand, the current search procedure of FDP requires that any solution must contain only one single action per step. Hence, solutions returned by FDP are optimal in terms of the number of actions.

Problem representation

FDP works on a structure that resembles the well-known GRAPHPLAN planning graph (Blum & Furst 1995). It is a leveled graph that alternates *propositions levels* and *actions levels*. The i-th propositions level represents the validity of the propositions at step i. The i-th actions level represents the possible values for the action that is applied at step i. Since FDP searches for optimal sequential plans, FDP structures do not contain no-ops actions.

Consistent FDP-structures

FDP makes use of consistency rules to remove from FDP-structures some values of proposition variables or actions that cannot occur in any valid plan. For example an action whose one precondition is not valid should not be considered, and then can be removed without loss of completeness. The search procedure maintains the consistency of the FDP-structure, so as to discard as soon as possible invalid litterals or actions. A consistent structure in which each action level contains a single action and such that the first proposition level corresponds to the initial state of the planning problem and the last level contains the goals, represents a solution plan.

FDP consistency rules are the following. A litteral l at level i is inconsistent (cannot be *true*) if one of the following situations hold:

1. **(forward persistency)**
   l is not true at level $i-1$ and no possible action at level $i-1$ has l as effect,
2. **(all actions delete)**
   any possible action at level $i-1$ deletes l,
3. **(backward persistency)**
   l is not true at level $i+1$ and no possible action at level i deletes l,
4. **(opposite always required)**
   any possible action at level i has $\neg l$ as precondition.

A possible action a at step i is inconsistent (cannot occurs) if one of the following situations hold:

1. **(falsified precondition)**
   a precondition of a is inconsistent at level i,
2. **(falsified effects)**
   an effect of a is inconsistent at level $i+1$,
3. **(effect required)**
   there exists a litteral l such that l is inconsistent at level i, $\neg l$ is inconsistent at level $i+1$, and l is not an effect of a.

Maintaining consistency

Making a FDP-structure consistent consists in removing inconsistent values and actions until none exists or a domain becomes empty. The mechanism is similar to arc consistency enforcing procedures in the domain of constraint satisfaction (Dechter 2003; Mackworth 1977). One major aspect of the procedure is that the removals are propagated forward and backward through the FDP-structure. Propagation stops with failure if a domain becomes empty and the procedure returns FALSE. In the other case the procedure stops with the consistent FDP-structure S.

Search procedure

To find an optimal plan, FDP starts with a one step FDP-structure, and extends it until a plan is found or a given fixed bound is reached. Each time the FDP-structure is extended, a depth-first search is performed. This ensures the optimality of the solution plan if one exists. FDP employs a *divide and conquer* approach to search for a plan of a given length: the structure is decomposed into smaller substructures and the procedure searches recursively each of them. The substructures are filtered so as to detect failures as soon as possible.

The decomposition mechanism currently performed is *splitting action sets*. It consists in partitionning the set of actions at a given step i so as to put together actions which have common deletions: The procedure searches for the undefined proposition variable p at step $i + 1$ for which the number of actions that delete it and the number of actions that do not are the closest. The FDP-structure is then decomposed into two substructures, one containing the actions at step i which delete p, the other containing the remaining actions at step i. The two substructures are then filtered.

When searching for a plan of length k, FDP uses a FDP-structure S:Initially each action set of S is set to A and each proposition variable is undefined. Then, the values which are not in the initial state and the opposites of the goals are removed and a preliminary filtering is performed on S. If S is inconsistent then the search stops with failure, there are no plans of length k. In the other case, FDP starts searching with the consistent structure S, which is decomposed into two substructures according to the splitting of an actions set. Nevertheless, the search procedure remains a depth first iterative deepening search, since it always chooses the first non singleton actions set for splitting, starting from the initial state. To produce each of the two substructures by actions set splitting, FDP just removes from the actions set the actions belonging to other actions subset. Then, each resulting substructure is filtered so as to remove inconsistent values and actions. If it is consistent, the search is recursively performed. These transformations continue until the (sub)structure becomes inconsistent or a valid plan.

Improving performance

FDP uses several techniques to avoid search efforts and then improve performance. They are: recording nogoods, evaluation of minimal plan length, avoidance of redundant actions sequences, elimination of literals and actions that are not relevant. These techniques are briefly discussed below.

Nogoods recording. Whenever the system produces a totaly defined state at a level i such that the recursive search from that state returns failure, this state and its distance to the golas are recorded as a nogood. Later, if the same state is reached but its distance to the goal step is less than or equal to the memorized distance, then there is no need to pursue the search. Recording nogoods improves drastically the performances of the search.

Minimal plan length. Anytime a propositional level F_i is completely instantiated, FDP performs a greedy evaluation of the length of a plan to achieve the goals from that state. It consists in choosing at each of the following steps the action which adds the most unsatisfied goals. In the best case these actions will constitute a valid plan. This heuristic is admissible: The number of steps needed to achieve the goals with this evaluation process cannot be greater than the number of steps actually needed in any valid plan. If at step k some goals are not achieved by the selected actions, then the search from the current state is aborted.

Redundant actions sequences. Since FDP searches sequential plans, it can generate equivalent permutations of "independent" actions and perform as many redundant processings. To avoid these useless processings, FDP discards the sequences of independent actions that do not verify an arbitrary total order on the actions denoted $\prec$.

Definition 1 (Ordered 2-Sequences) *The actions a_1 and a_2 are* independent *if the following situations hold:*

1. *no precondition of a_1 is an effect of a_2 and no precondition of a_2 is an effect of a_1*[1]*,*
2. *no deletion of a_2 is a precondition of a_1 and no deletion of a_1 is a precondition of a_2.*

The sequence (a_1, a_2) is an ordered 2-sequence *if either a_1 and a_2 are independent and $a_1 \prec a_2$, or a_1 and a_2 are not independent.*

FDP discards unordered 2-sequences. Besides, it also discards sequences whose actions have exactly opposite effects, as such sequences are useless in a plan.

To avoid sequences that do not verify the order, the following rules are added to the definition of inconsistent actions:

4. **(no backward ordered 2-sequence)**
a is inconsistent at level i if there exists no action a' at level $i - 1$ such that (a', a) is an ordered 2-sequence,
5. **(no forward ordered 2-sequence)**
a is inconsistent at level i if there exists no action a' at level $i + 1$ such that (a, a') is an ordered 2-sequence.

Relevant literals and actions. FDP searches optimal sequential plans. Then actions which do not help effectively to achieve the goals are useless and should not be considered. Basically relevant actions are the ones which add goals at the

[1]If a_1 requires a fact which is added by a_2, it is possible in some situations that the sequence (a_2, a_1) must be authorized. Then a_1 and a_2 should not be considered as independent.

last level. This property can be propagated backwards iteratively introducing the notion of *relevant literals and actions* at some steps:

1. a literal l is relevant at level i if there exists an action a at level i such that l is a precondition of a and a is relevant at level i,
2. an action a is relevant at level i if one of its effects is relevant at level $i + 1$.

At any moment during the search, actions that are not relevant at a given level can be removed from this step as it could not serve in any minimal solution.

Mutually exclusive propositions and actions FDP does not implement any specific processing for mutual exclusion relations, in particular those handled in GRAPHPLAN. Indeed, they are useless since FDP produces only sequential plans, and the effects of mutual exclusions of propositions are redundant with FDP inconsistency rules.

Conclusion and perspectives

Compared to other optimal sequential planners FDP seems to be competitive. Its advantage is its regularity: maintaining consistency, memorizing invalid states, and discarding redundant sequences, in addition with a fast and light search procedure, let FDP quickly detect deadends.

Its consistency rules and its decomposition strategies allow to operate backward chaining search or bidirectional search and more generally undirectional search. FDP could be improved with other evaluations of the minimal distance to the goals (Haslum, Bonet, & Geffner 2005) and concurrent bidirectional searches which could cooperate through valid or invalid states. The lack of termination criterion will be also addressed in future work. Finally FDP could be extended to handle valued actions and to compute plans of minimal costs. Also, planning with ressource will be a matter of development.

References

Baioletti, M.; Marcugini, S.; and Milani, A. 2000. Dpplan: An algorithm for fast solutions extraction from a planning graph. In *AIPS*, 13–21.

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.

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

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In Veloso, M. M., and Kambhampati, S., eds., *AAAI*, 1163–1168. AAAI Press AAAI Press / The MIT Press.

Korf, R. 1985. Macro-operators: A weak method for learning. *Artificial Intelligence* 26(1):35–77.

Lopez, A., and Bacchus, F. 2003. Generalizing graphplan by formulating planning as a CSP. In Gottlob, G., and Walsh, T., eds., *IJCAI*, 954–960. Morgan Kaufmann.

Mackworth, A. 1977. Consistency in networks of relations. In *Artificial Intelligence*, 8:99–118.

Rintanen, J. 1998. A planning algorithm not based on directional search. In *KR*, 617–625.

Fast (Diagonally) Downward

Malte Helmert
Institut für Informatik, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee, Gebäude 052, 79110 Freiburg, Germany
helmert@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 tasks in a hierarchical fashion.

Fast Downward won the propositional satisficing track of the 4th International Planning Competition (IPC4). At the 5th International Planning Competition (IPC5), a (mostly) unchanged version of the planner was entered to provide a reference point for comparing to the earlier state of the art.

Introduction

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). Fast Downward itself is described in much detail in a JAIR article (Helmert 2006). For this reason, we only provide a very brief overview in the following.

Structure of the planner

Fast Downward consists of three separate programs:

1. the *translator* (written in Python),
2. the *knowledge compilation* module (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.

Translator

The purpose of the *translator* is to transform the planner input, specified in the propositional fragment of PDDL (including ADL features and derived predicates, but not the preferences and constraints introduced for IPC5), into a multi-valued state representation similar to the SAS$^+$ formalism (Bäckström & Nebel 1995).

The main components of the translator are an efficient grounding algorithm for instantiating schematic operators and axioms and an invariant synthesis algorithm for determining groups of mutually exclusive facts. Such fact groups are consequently replaced by a single multi-valued state variable encoding *which* fact (if any) from the group is satisfied in a given world state, rather than encoding the truth of each fact in a separate state variable.

The translator can be used independently from the rest of the planner, and has already proved to be a useful component in planning systems not related to Fast Downward (van den Briel, Vossen, & Kambhampati 2005).

Knowledge Compilation

Using the multi-valued task representation generated by the translator, the *knowledge compilation* module is responsible for building some data structures which play a central role in Fast Downward's search engine.

First and foremost, it determines the *causal graph* of the input task, whose purpose is to encode the information which state variables are relevant to changing which other state variables of the task. Together with *domain transition graphs* for each state variable, which encode the ways in which a given state variable may change its value through operator applications, it forms the basis for the recursive computation of the CG heuristic.

The knowledge compilation module also generates *successor generators* and *axiom evaluators*, data structures for efficiently determining the set of applicable actions in a given state of the planning task and for evaluating the values of derived state variables.

Search Engine

Using these data structures, the *search engine* attempts to find a plan using greedy best-first search with some enhancements such as the use of *preferred operators* (similar to helpful actions in FF) and *deferred heuristic evaluation*, which mitigates the impact of large branching factors in planning tasks with accurate heuristic estimates.

The search engine can also be configured to use several heuristic estimators (namely, the CG heuristic and the FF heuristic) in tandem within an algorithm called *multi-heuristic best-first search*. This search algorithm attempts to exploit strengths of the utilized heuristics in different parts of the search space in an orthogonal way. The planner configuration using multi-heuristic best-first search is called *Fast Diagonally Downward*, because it combines the "downward" thrust of the CG heuristic with the "forward" thrust of the FF heuristic.

Fast Downward also includes a third search algorithm called *focused iterative-broadening search*, but as of this writing, it is not clear whether or not this algorithm will be used for the IPC5 benchmark set.

New Developments

Since IPC4, apart from some bug fixes, we made only one modification to Fast Downward.

In some planning domains, it is clear from the multi-valued task description that some state variables can change their value in essentially arbitrary ways without further conditions on other state variables. For example, state variables which encode vehicle locations in transportation domains such as LOGISTICS or DEPOTS never have causal dependencies on other state variables in the task (i. e., they are source nodes in the causal graph) and can never assume a value from which the initial value cannot be restored anymore (i. e., their domain transition graphs are strongly connected).

In tasks where such state variables are present, Fast Downward can now apply *safe abstraction* to remove the state variables from the planning task altogether and plan in the resulting abstract planning task. After planning, the necessary actions to convert the generated abstract plan to the concrete level can be inserted in a straight-forward manner.

Safe abstractions can substantially speed up planning; indeed, in very simple cases like the LOGISTICS domain, repeated application of safe abstraction can completely solve the planning task, without requiring any search. However, there is often a price to pay in plan quality.

At IPC5, we have run Fast Downward both in a version that does use safe abstractions where possible and in a version which never uses them. The latter version of the planner is identical to Fast Downward at IPC4 (modulo bug fixes) and thus serves as a useful reference point.

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS$^+$ planning. *Computational Intelligence* 11(4):625–655.

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

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proc. ICAPS 2004*, 161–170.

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

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

van den Briel, M.; Vossen, T.; and Kambhampati, S. 2005. Reviving integer programming approaches for AI planning: A branch-and-cut framework. In *Proc. ICAPS 2005*, 310–319.

New Features in SGPlan for Handling Preferences and Constraints in PDDL3.0[*]

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
{*chsu,wah*}*@manip.crhc.uiuc.edu*

Ruoyun Huang and Yixin Chen
Department of Computer Science and Engineering
Washington University in St Louis
St Louis, MO 63130, USA
rh11@cec.wustl.edu
chen@cse.wustl.edu

Abstract

In this paper, we describe our enhancements incorporated in SGPlan (hereafter called SGPLan$_5$) for supporting the new features of the PDDL3.0 language used in the Fifth International Planning Competition (IPC5). Based on the architecture of SGPlan that competed in the Fourth IPC (hereafter called SGPLan$_4$), SGPLan$_5$ partitions a large planning problem into subproblems, each with its own subgoal, and resolves those inconsistent solutions using our extended saddle-point condition. Subgoal partitioning is effective for solving large planning problems because each partitioned subproblem involves a substantially smaller search space than that of the original problem. In SGPLan$_5$, we generalize subgoal partitioning so that the goal state of a subproblem is no longer one goal fact as in SGPLan$_4$, but can be any fact with loosely coupled constraints with other subproblems. We have further developed methods for representing a planning problem in a multi-valued form in order to accommodate the new features in PDDL3.0, and for carrying out partitioning in the transformed space. The multi-valued representation leads to more effective heuristics for resolving goal preferences and trajectory and temporal constraints.

DESIGN GOALS

SGPLan$_5$ has participated in the suboptimal track of the deterministic part of the Fifth International Planning Competition (IPC5). It was designed to fully support the PDDL3.0 language (Gerevini & Long 2005) specifications.

PDDL3.0, the modeling language used in IPC5, extends the previous PDDL2.2 (Edelkamp & Hoffmann 2004) specifications by introducing several new features: a) simple preferences over only action preconditions or goals, b) qualitative preferences that are logical preferences over trajectory constraints, c) complex constraints that are trajectory constraints with metric time and possibly numeric fluents, and d) complex preferences that are preferences over trajectory constraints with metric time and possibly numeric fluents.

SGPLan$_5$ uses a *multi-valued domain formulation (MDF)* based on the SAS+ formalism. MDF has been used by other planning systems, such as Fast Downward (Helmert 2004) and IP planner (van den Briel, Vossen, & Kambhampati 2005). We have implemented our own preprocessing engine for translating a PDDL3.0 problem into MDF. There are several reasons for using MDF.

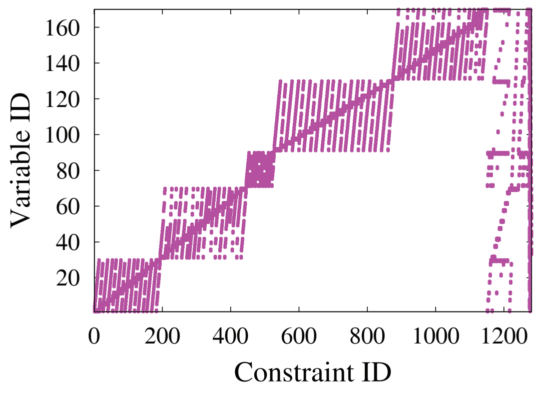

Figure 1: The locality of the constraint-variable structure in the fifth instance of the *TPP-SimplePreferences* domain.

a) MDF provides a more compact representation than a binary-valued representation and leads to a more effective partitioning of the constraints.
b) Using transition graphs, MDF facilitates the analysis of the causal dependencies among variables. The analysis leads to much more accurate heuristic guidance values than those of the Metric-FF heuristic used in SGPLan$_4$.
c) Using the new heuristic function, high-quality approximate plans can be found for resolving temporal constraints in PDDL3.0 problems efficiently.

CONSTRAINT LOCALITY

Constraint partitioning in SGPLan$_5$ is based on the locality of constraints observed in IPC5 planning domains. Figure 1 illustrates the constraint-variable structure in the fifth instance of the *TPP-SimplePreferences* domain. Each variable represents an action and its schedule in the plan, and a constraint can be a mutual exclusion, an inconsistent state

[*]Research supported by the National Science Foundation Grant IIS 03-12084.

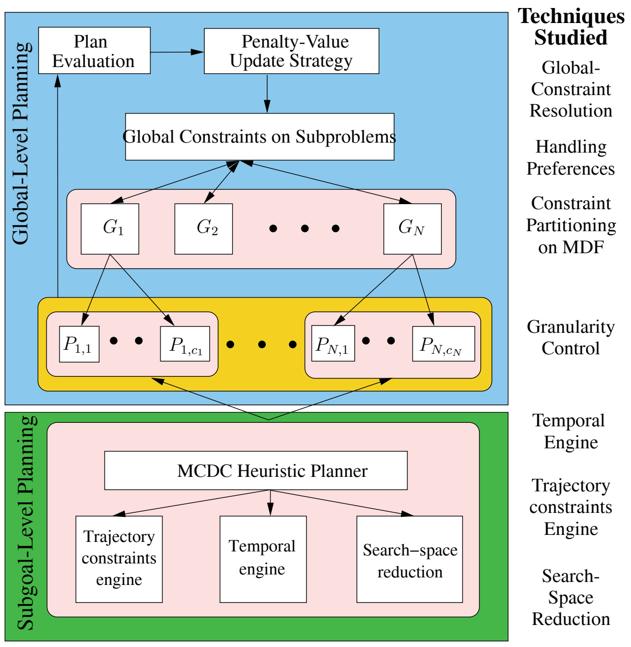

Figure 2: Architecture of SGPLan$_5$.

variable assignment, or a trajectory constraint. It is obvious that a majority of the constraints can be localized after identifying those (global) constraints with variables that have strong causal dependencies to many other variables.

We have developed SGPLan$_5$ to exploit constraint locality in all IPC5 domains. However, constraints in PDDL3.0, both hard and soft, can be over the intermediate states of a plan, in addition to those hard constraints on the final states as in PDDL2.2. Hence, we must extend SGPLan$_4$ (Chen, Wah, & Hsu 2006) that only aims to satisfy a conjunctive list of the conditions on the final states. In SGPLan$_5$, we have extended our partitioning approach based on MDF. The multi-valued domain analysis allows us to eliminate a number of mutual exclusions as well as inconsistencies among the soft constraints before the constraints are partitioned.

Note that, although constraint locality is common in all IPC5 benchmarks, the difficulty of resolving the constraints varies across domains. For instance, we have found that all the subproblems in the *OpenStacks* domain are trivial to solve, but the major challenge is to enforce the consistency of its shared variables.

ARCHITECTURE OF SGPLan$_5$

By formulating a subproblem in such a way that each has one goal state, SGPLan$_5$ partitions a planning problem into subproblems and finds a feasible plan for each subgoal (Figure 2). In the global level, it partitions the problem by its multi-valued state variables and resolves its violated global constraints using the theory of extended saddle points (Wah & Chen 2006). In the local level, it calls a basic planner for solving each partitioned subproblem, using the violated global constraints and the global preferences as biases.

Global-Level Search

Partitioning Strategy. We have observed that many constraints have a strong locality if we can identify those constraints that involve many state variables. From the causal graph, we can extract those (low-level) state variables that influence many other state variables. Dependencies due to these state variables would cause active mutual exclusions across subproblems no matter how the constraints are partitioned. On the other hand, there are (high-level) state variables whose state transitions require a set of other (low-level) variables. Since constraint locality is associated with high-level state variables, we can formulate constraints that involve state variables across partitions as global constraints. Also, we have chosen an optimal grain size that minimizes the number of shared variables in order to reduce the number of global constraints.

Resolution of Global Constraints. A planning problem solved by SGPLan$_5$ is defined in mixed space with a nonlinear objective and one or more constraints that may be procedural. SGPLan$_5$ implements a search to find extended saddle points (ESPs) of a penalty function derived from the problem (Wah & Chen 2006), where the penalty function consists of the sum of the objective and the transformed constraint functions weighted by penalties, and an ESP is a local minimum of the penalty function with respect to the original variables and a local maximum with respect to the penalties. The search algorithm is based on the extended saddle-point condition (ESPC) that shows the one-to-one correspondence between the ESPs and the feasible local optima.

An important property of the ESPC is that it is true for all penalty values larger than a minimum threshold. This property allows the search of points that satisfy the ESPC to be found iteratively, with an inner loop that looks for a local minimum of the penalty function, and an outer loop that looks for any penalty values larger than the threshold. The property also allows the search of ESPs to be partitioned into multiple searches, each looking for a local ESP in a subproblem, and an outer loop that resolves the inconsistencies among the subproblems.

A direct implementation of the ESPC in a search algorithm may get stuck in an infeasible region when the objective is too small or when the penalty values and/or constraint violations are too large. To address this issue, SGPLan$_5$ performs backtracking to escape from infeasible local traps.

Handling Constraints by the MCDC Heuristic. Since the *minimum causal dependency-cost* (MCDC) heuristic can generate a highly accurate approximate plan for each state, we can obtain a tight lower bound on the estimated makespan from each state and use it to eliminate the state when temporal constraints are violated. For temporal constraints in the form of deadlines, we prune a state whose MCDC value exceeds the deadlines. For side trajectory constraints, we prune a state whose approximate MCDC plan violates the constraints.

Handling Preferences. We have classified all trajectory preferences into two categories.

The first class of preferences consists of those soft constraints on the final state and the persistent soft constraints (model operator *always*). We consider them with the original goal definitions because they have temporal overlaps on the final state. Although it is not easy to find an optimal set of soft constraints to be satisfied, it is trivial to compute their violation cost, when given an assignment of all state variables involved in the goal preferences. Therefore, we enumerate all reachable elements of each state variable involved and choose an optimal combination of facts to achieve. The enumerations can be highly decomposed because those constraints on the final state also have strong localities. It is still possible to make an unreachable assignment, even though the MDF analysis can detect many implicit mutual exclusions. For those unreachable assignments, we perform backtracking to find alternative assignments. When the cost of the assignment (such as in the form of a weighted sum of preference violations and plan quality) is unknown until planning ends, we also perform backtracking to find better solutions after a plan has been found.

The second class of preferences are those with insufficient information on their satisfiability. This may happen because the related soft constraints are not always active. To address this issue, we have devised a *relax-and-tighten* strategy that initially ignores all those preferences belonging to this class and that penalizes those unsatisfied preferences in order to generate a solution. As is done earlier in resolving constraints, we have developed a number of heuristics for estimating the reachability of preferences and have applied iterative refinements until no better solutions can be found.

Local-Level Basic Planner

Our basic planner follows the heuristic search algorithm used in Metric-FF (Hoffmann 2003), but employs a new heuristic based on the multi-valued formulation.

MCDC Heuristic Planner. Using the multi-valued formulation, we have implemented a new search heuristic by exploring the value transition graph of each variable and the causal dependencies between the transition graphs. The general idea is inspired by and similar to the heuristic used in the Fast Downward planner (Helmert 2004). However, our MCDC heuristic is very different from the Fast Downward heuristic in a number of aspects.

First, our MCDC heuristic employs a recursive depth-first search for generating a heuristic plan with the minimum cost without pruning the causal graphs. In contrast, the Fast Downward heuristic is incomplete since it performs strongly-connected-component analysis and removes nodes with low connectivity. We have found that our complete recursive search leads to much less node expansions.

Second, we have developed a set of strong necessary conditions for pruning infeasible or dominated paths when searching for the best approximate plan. We have also developed an algorithm for detecting symmetric objects in a given state to further reduce the cost of the MCDC heuristic calculations. These pruning rules can reduce the computing time of our heuristic by one to two orders of magnitude.

Third, in addition to sequential propositional planning supported by the Fast Downward heuristic, MCDC supports parallel temporal planning and can generate estimates of makespans for temporal plans.

The heuristic plan found by MCDC is approximate because the transition graphs found are not complete and some of its actions may not be supported. Moreover, numerical and trajectory constraints are ignored in MCDC.

MCDC is not admissible because, when computing an approximate plan, we consider each subgoal individually and add up the costs of all subgoals in order to estimate the overall heuristic value. Thus, MCDC ignores the positive interactions among the subgoals and is not admissible.

Search-Space Reduction. Before solving a partitioned subproblem, we can often eliminate many irrelevant actions in its search space. We identify those relevant actions by traversing the causal graphs in MDF and by ignoring actions that are not useful for achieving the current subgoal state variables. We also prioritize actions that do not cause an inconsistent assignment of multi-valued state variables. This is done by following our partitioning setting to compute a set of local state variables for each subproblem, and by applying the helpful action idea introduced in FF (Hoffmann & Nebel 2001) to defer those actions that change the value of shared state variables.

References

Chen, Y. X.; Wah, B. W.; and Hsu, C. W. 2006. Temporal planning using subgoal partitioning and resolution in SGPlan. *J. of Artificial Intelligence Research*.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classic part of the 4th International Planning Competition. Technical report, Tech. Rep. 195, Institut für Informatik, Freiburg, Germany.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences for PDDL3. Technical report, R.T. 2005-08-07, Dept. of Electronics for Automation, U. of Brescia, Brescia, Italy.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *ICAPS*, 161–170.

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

Hoffmann, J. 2003. The Metric-FF planning system: Translating ignoring delete lists to numeric state variables. *J. of Artificial Intelligence Research* 20:291–341. `http://www.mpi-sb.mpg.de/~hoffmann/metric-ff.html`.

van den Briel, M.; Vossen, T.; and Kambhampati, S. 2005. Reviving integer programming approaches for AI planning: A branch-and-cut framework. In *ICAPS*, 310–319.

Wah, B., and Chen, Y. X. 2006. Constraint partitioning in penalty formulations for solving temporal planning problems. *Artificial Intelligence* 170(3):187–231.

OCPlan – Planning for soft constraints in classical domains

Bharat Ranjan Kavuluri bharat@cs.iitm.ernet.in
Naresh Babu Saladi snaresh@cse.iitm.ernet.in
Rakesh Garwal rakesh@cse.iitm.ernet.in
Deepak Khemani khemani@iitm.ac.in
Department of Computer Science and Engineering
Indian Institute of Technology Madras Chennai-36, India

Introduction

Recent research in AI Planning is focused on improving the quality of the generated plans. PDDL3 [Alfonso and Gerevini 2005] incorporates hard and soft constraints on goals and the plan trajectory. Plan trajectory constraints are conditions that need to be satisfied at various stages of the plan. Soft goals are goals, which need not necessarily be achieved but are desirable. To deal with these, we use an extension of Constraint Satisfaction Problem (CSP), called Optimal Constraint Satisfaction Problem (OCSP) [Sachembacher and Williams 2005]. OCSP has allowance for defining soft constraints. Each soft constraint is associated with a penalty, which will be levied if the constraint is violated. The solver arrives at a solution that minimizes the total penalty (Objective function) and satisfies all hard constraints. In this system, we introduce an OCSP encoding for the classical planning problems with plan trajectory constraints, soft and hard goals. Modal operators are handled by preprocessing and imposing new constraints over the existing GP-CSP encoding [Minh Binh and Khambampati 2001]. The obtained OCSP is solved using A* search and forward checking [Vipin Kumar 1992]. Figure 1 describes the overall planning process.

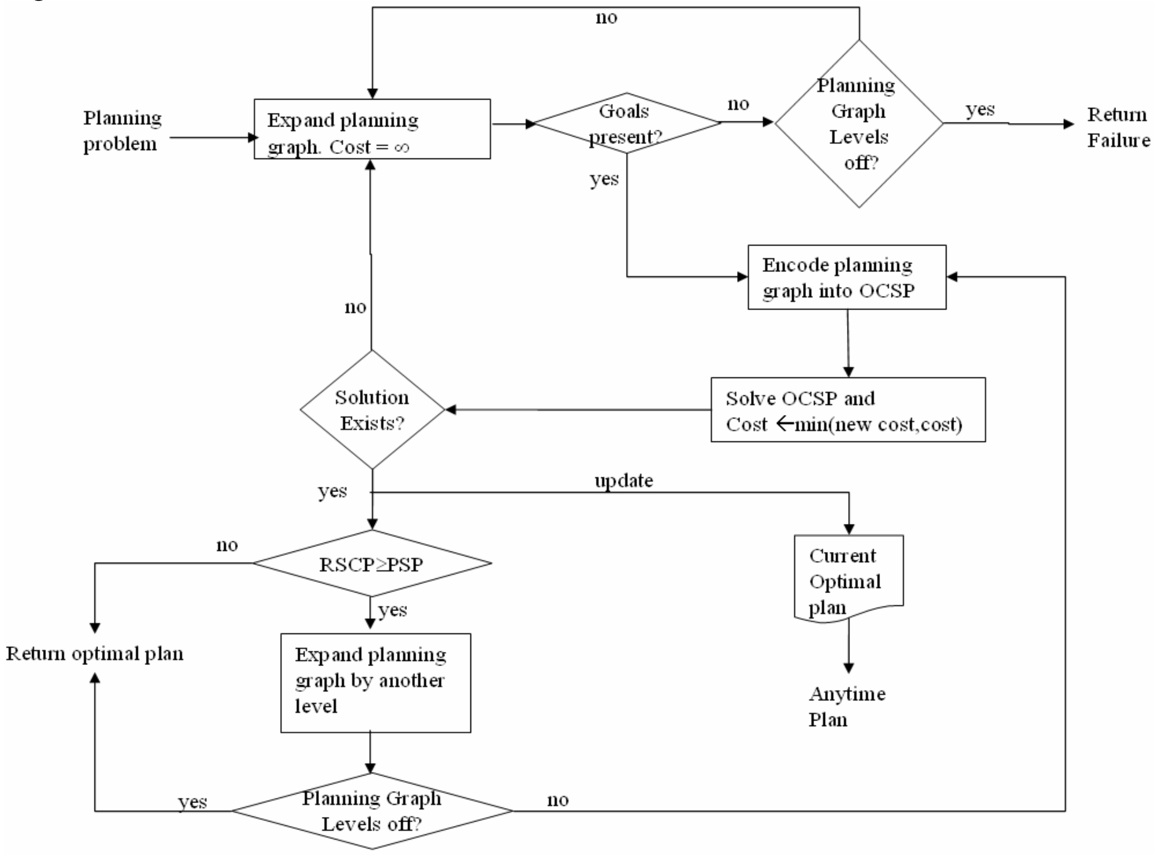
Planning problem
Expand planning graph. Cost = ∞
Goals present?
no
Planning Graph Levels off?
yes
Return Failure
no
yes
Encode planning graph into OCSP
Solve OCSP and Cost ←min(new cost,cost)
Solution Exists?
no
yes
update
Current Optimal plan
Anytime Plan
RSCP≥PSP
no
yes
Return optimal plan
Expand planning graph by another level
Planning Graph Levels off?
yes
no

Figure 1 The overall planning process

The overall planning process

Initially, the planning graph is expanded until all the required predicates appear in various predicate levels and there is a non-mutex goal set available for the hard goals. At this stage, the planning graph is encoded to OCSP and passed to the solver. Until there is no solution satisfying all the constraints, the planning graph is expanded, encoded into OCSP and checked for a solution. This loop continues until either the solution for the hard constraints is found or the planning graph levels off (in which case the planning process is abandoned and failure is announced). If there is a solution satisfying all hard constraints, then the current optimal plan is updated along with its cost and the system checks whether the penalty incurred due to the remaining soft constraints (RSCP, Remaining Soft Constraints Penalty) is greater than the plan step penalty (PSP). If that is the case (RSCP ≥ PSP), then the system looks to satisfy the remaining soft constraints by expanding the planning graph by one more level. This loop continues until either the planning graph levels off or RSCP ≤ PSP. The optimal plan is returned at the termination of this loop. The PSP is obtained from the problem file. If PSP is not specified in the problem file, then it is taken as zero. In this case, the planner will try to improve the plan until the soft penalty is zero or the planning graph levels off. As the OCSP can only deal with unary soft constraints, the plan step penalty cannot be embedded into the OCSP directly and hence it is taken care in the overall planning process. The OCSP solution method is shown in Figure 2.

OCSP solution method

Each soft constraint in the OCSP is associated with a decision variable. Solution to OCSP contains assignment to decision variables as well as non-decision variables. We follow a hybrid algorithmic approach for solving the generated OCSP. This algorithm exploits the distinction between decision variables (which determine the valuation of an assignment) and non-decision variables (which determine only the consistency of an assignment). The algorithm uses forward checking for solving hard constraints and A* search [Hart, Nilsson, & Raphael 1968] for solving soft constraints. The flow chart for this method is shown in figure 2. In this method, initially, the algorithm performs an A* search over the assignments to the decision variables. If, all the possible assignments to the decision variables in A* search result in a penalty that is more than the cost calculated in the previous iteration of the overall planning method (see figure 1), it returns the stored optimal solution. Otherwise, A* algorithm passes the best possible assignment for the decision variables to the second step in the OCSP solution process in which it solves for the non-decision variables using forward checking [Vipin Kumar 1992]. If a solution is not found to the non-decision variables, A* search is again invoked. This process continues until either a solution that satisfies all hard constraints or all possible assignments to the decision variables are exhausted.

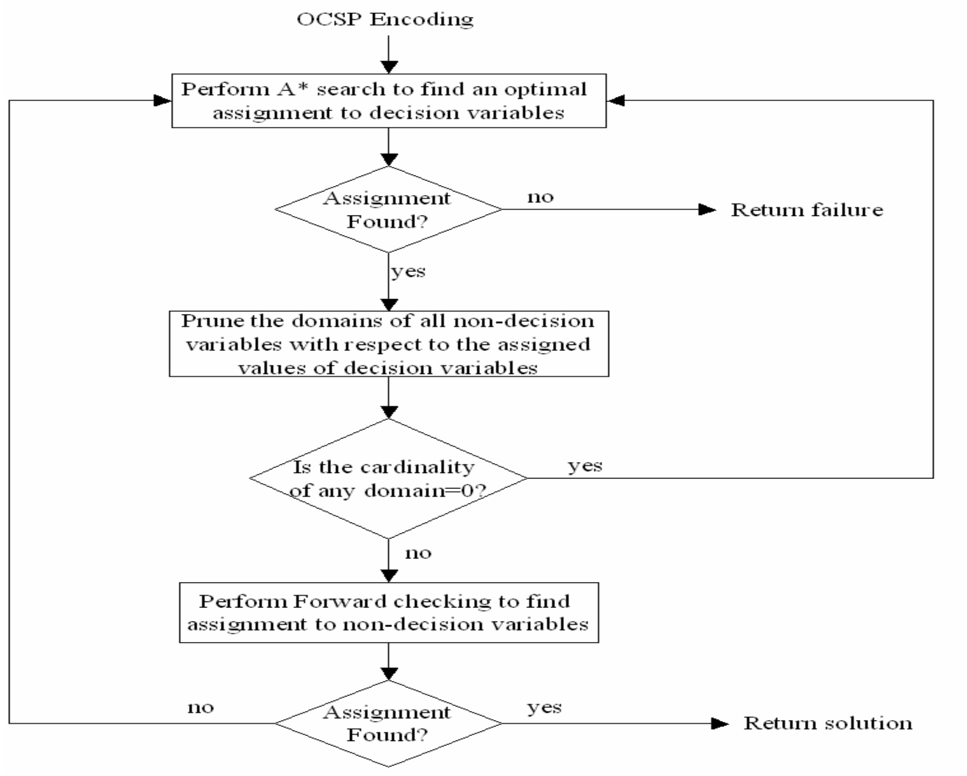
OCSP Encoding
Perform A* search to find an optimal assignment to decision variables
Assignment Found?
no
Return failure
yes
Prune the domains of all non-decision variables with respect to the assigned values of decision variables
Is the cardinality of any domain=0?
yes
no
Perform Forward checking to find assignment to non-decision variables
Assignment Found?
no
yes
Return solution

Figure 2 The OCSP solution Method

References

Alfonso Gerevini and Derek Long. August 2005. Plan Constraints and Preferences in PDDL3. Technical Report, Department of Electronics for Automation, University of Brescia, Italy.

Hart, P.E., Nilsson, N. J., and Raphael, B. "A formal basis for the heuristic determination of minimum cost paths", IEEE Transactions on Systems Science and Cybernetics, SSC-4(2), 100-107 (1968).

Martin Sachenbacher and Brian C. Williams. 2005. Solving Soft Constraints by Separating Optimization and Satisfiability. 7th International Workshop on Preferences and Soft Constraints, held in conjunction with 11th International Conference on Principles and Practice of Constraint Programming.

Minh Binh Do and Subbarao Kambhampati. 2001. Planning as Constraint Satisfaction: Solving the Planning Graph by Compiling it into CSP. Artificial Intelligence 132(2): 151-182.

Vipin Kumar. 1992. Algorithms for constraint satisfaction problems: A survey. AI magazine, 13(1):32-44.

SATPLAN04: Planning as Satisfiability

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

Bart Selman
Department of Computer Science
Cornell University
Ithaca, NY 14853 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-interferring) 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 disjuction 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 (4) while non-skinny encodings include both (4) and (5).

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 Fraiser University under the direction of Prof. David Mitchell. Linux binaries of siege can be downloaded from `http://www.cs.sfu.ca/ loryan/personal/`.

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 significantly 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.

Acknowledgements

We thank Lawrence Ryan for permission to incorporate siege in SATPLAN04, and Joerg Hoffmann for contributing code.

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.

The resource YAHSP planner

Marie de Roquemaurel*, Pierre Régnier* and Vincent Vidal$^+$

* IRIT - Université Paul Sabatier
118, route de Narbonne
31062 Toulouse, cedex 9, France
{deroquemaurel, regnier}@irit.fr

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

Introduction

The rYAHSP planner (resource Yet Another Heuristic Search Planner) implements the management of resources in the YAHSP planner (Vidal 2004) who got a 2nd place in the suboptimal STRIPS category of the IPC'04 competition.

Many state-space heuristic planners work in the classical framework such as FF (Hoffmann & Nebel 2001) and YAHSP (Vidal 2004) or in the numerical one such as Metric-FF (Hoffmann 2003), SAPA (Do & Kambhampati 2003) and TP4 (Haslum & Geffner 2001). FF, YAHSP and Metric-FF use a relaxed planning graph to compute a heuristic that estimates the distance to the goal. This distance is obtained by building a planning graph from the current state and from a relaxation of the original task. The heuristic value is the number of actions of the relaxed solution-plan extracted from this graph.

One can observe that the actions contained in the relaxed plans are generally contained in the solution-plan of the original problem. Using this observation, YAHSP uses these actions in a lookahead strategy to try to build a valid plan. If this plan exists, it leads to a state that is potentially closer to the goal. This state is then treated as a direct descendant of the current state by a complete best-first search algorithm. Moreover, YAHSP improves the classification, introduced by FF, of the applicable actions in helpful actions and rescue actions.

rYAHSP adapts the strategy of YAHSP to the numeric framework, taking into account the plan metric only if it can be transformed into additive action cost minimization. rYAHSP can parse and plan with PDDL domains with :fluents and :typing requirements.

After some definitions, we present the computation of the heuristic that uses a numeric relaxed planning graph. Finally, we describe the lookahead strategy used within the numeric framework.

Preliminary definitions

A *numeric state* S is a finite set of propositions (noted *prop(S)*) and a finite set of numeric variables with values (noted *varNum(S)*). A *numeric action* a is a pair $< prec(a), eff(a) >$ where *prec(a)* is a set of propositional and numeric preconditions of a and *eff(a)* is a set of propositional and numeric effects of a. *add(a)*, *del(a)* and *effNum(a)* respectively denote add effects, del effects and the numeric effects of a.

A proposition p is *satisfied* in a state S iff $p \in S$. A numeric condition c is *satisfied* in a state S iff the numeric variables contained in c have a value defined in S and verify c. A set of propositions and conditions is *satisfied* in a state S iff all its propositions and conditions are satisfied in S.

A numeric variable of an action a is *updated* when its value is updated by the effects of a. *updated(a)* will represent the set of variables updated by a.

A *numeric planning problem* is a triple $<A,I,G>$ where A is a set of numeric actions, I is an initial numeric state and G is a set of propositions and numeric conditions. The *application* of an action a on a state S (noted $S \uparrow a$) is possible iff $prec(a)$ is satisfied in S. The resulting state S' is defined by $prop(S') = (prop(S) - del(a)) \cup add(a)$. $varNum(S')$ contains numeric variables associated with the values resulting of the application of all numeric effects of a. If a numeric variable is not updated by a, it remains unchanged.

A *plan* is a sequence of actions. Let $P = <a_1, a_2, \ldots, a_n>$ be a plan. P is a *valid plan* 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 $S_n = ((((S \uparrow a_1) \uparrow a_2) \ldots) \uparrow a_n)$. In that case, S_n is said to be *reachable* from S for P and P is a *solution-plan* if G is satisfied in S_n.

The quality of a plan is estimated by a function called *metric*[1] of the plan. This function is a non temporal arithmetic expression related to numeric variables. In this article, we will restrict ourselves with the linear metric that we will attempt to minimize.

Computation of the heuristic

In this chapter, we describe the relaxation of the numeric problem, the construction of the relaxed numeric graph, the extraction of a relaxed solution-plan and the computation of the heuristic used in rYAHSP.

Relaxation of a numeric problem

rYAHSP uses the relaxation of a numeric problem introduced by Metric-FF (Hoffmann 2003). For the relaxed problem, the termination of the construction of the planning

[1]This metric is expressed in the field ":metric" introduced into language PDDL 2.1 (Fox & Long 2003).

graph imposes *monotonous* numeric constraints. A constraint c is said to be monotonous if, when c is satisfied in the state S, it is satisfied in all the states S' in which all the numerics variables have a value equal or greater than their value in S.

If the deletes and the effects that decrease the value of a variable are ignored for all the actions, then this property is verified (Hoffmann 2003). However, this type of relaxation restricts the use of the language to only increasing, decreasing and assignments. Moreover, the preconditions, the effects and the metric of the plan will only use arithmetic numeric expressions.

The *relaxed application* of an action a in a numeric state S ($S \uparrow^r a$) is thus carried out by ignoring the deletes of the action and the numeric effects that decrease the value of a numeric variable. The resulting state $S' = S \uparrow^r a = (prop(S'), varNum(S'))$ is such that :

- $prop(S') = prop(S) \cup add(a)$,
- $varNum(S')$ contains the numeric variables resulting from the application of the only effects of a that increase their values : $\forall\ u \in updated(a), \forall\ e \in effNum(a)$, *if val(u, e, S) > val(u, S) then e is ignored*[2].

Expansion of the relaxed graph

The planning graph is incrementally built from a state S and the relaxed application of the actions of A. At each level, we add all the add effects of the applicable actions. For each numeric variable v, we calculate :

- the sum of the values of all the increasing of v in this level,
- the maximum value of all the assignments of v.

We affect to the variable the maximum value that it could reach. This value is the maximum of the two preceding values.

The construction of the relaxed graph stops on a success if all the goals are satisfied, or fails if they are not and a fixed point is reached, i.e. :

- if there is no new proposition in the level $n + 1$ that was not in the level n,
- if, for all the numeric variables, either their maximum remains the same in the level n and in the level $n + 1$, or the maximum reaches the maximum value required by all the preconditions of the actions of the problem.

We will consider the framework of acyclic assignments. (Hoffmann 2003) demonstrates that in this framework, if the expansion of the graph fails from a state S, then there is no relaxed solution-plan from S.

Extraction of a solution and computation of the heuristic

When the construction of the relaxed graph is a success, the graph is pruned, starting from the goals. We preserve all the actions establishing these goals, and their preconditions become the new goals to solve. This process continues until the initial level is reached.

Starting from the last level of the reduced graph, we then try to extract a relaxed solution-plan using a backward search algorithm. During the insertion of an action a in the current relaxed plan, the set G of the current goal g at level n is updated with the effects and the preconditions of a :

1. The additions of a included in G are deleted from G because they are now satisfied.
2. G is updated with the assignment effects of a :
   - If the value obtained by the assignment of a variable is lower than the value required by a goal g at level n, then this effect is ignored,
   - else, the goal g at level n is deleted. We then add in G the fact that all numeric variables in the expression of the affected variable must have a value required at level $n - 1$ higher or equal to their maximum value at level $n - 1$.
3. If a increments the variable value of a goal g at level n, then the required value in g in the same level is decreased by the increment.
4. The propositional preconditions of a not included in G are added in G. If a precondition was required in G at level n, then it becomes required at level $n - 1$.
5. For each numeric precondition of a, we add in G, at level n, the fact that all numeric variables in the precondition must have a value higher or equal to their maximum value at level n.

During the extraction of the graph, the quality of the relaxed solution-plan depends on the choice and on the scheduling of the actions that establish the goals. To choose an action a among the applicable actions that solve a goal g, rYAHSP improves this quality combining several criteria :

1. The minimization of the number of actions of the plan :
   - preferring the noop [3],
   - preferring an action satisfying g thanks to an assignment, rather than a set of actions satisfying g by increasing.
2. The influence of the action on the metric, calculating the difference between the metric in the current state S and the state $S' = S \uparrow a$.
3. The choice of the least expensive action, the cost of an action being a function of the levels of apparition of its (propositionals and numerics) preconditions in the graph.

An ad-hoc combination of these criteria is used to extract a solution-plan of the relaxed graph. The cost of this solution-plan is then used as heuristic.

Lookahead strategy

Let $< A, I, G >$ be a planning problem and S, S' be two states. Let P be a valid plan which, applied to the state S,

[2]Let u be a numeric variable, S be a state, and e be an effect, $val(u, S)$ indicates the value of u in S. $val(u, e, S)$ indicates the value of u after the application of e in S

[3]The *noop* of a proposition p is an action that have only p as precondition and effect to add.

leads to the state S' reachable from S that brings closer to the goal. The state S' is said to be a *lookahead* state and it is considered by the search algorithm as a direct descendant of S. P is said to be a *lookahead* plan.

One can observe that the actions contained in the relaxed plans are generally contained in the solution-plan of the original problem. These actions can thus be firstly used to find a lookahead plan. Taking that into account, we use as much as possible relaxed plan actions. When no action of the relaxed plan is applicable, we replace one action by another, selected in the set of the problem actions.

The construction of the lookahead plan LP, starting from a state S and from a relaxed plan RP, is executed in two steps :

1. Each time that an action a of RP is applicable to S, a is added at the end of LP. S is updated by applying a. The actions that cannot be applied are collected in a list named *failure*. When RP is entirely traversed, it is updated with the *failure* list, and this process is repeated until RP is empty or when no more action can be added to LP.
2. When no action can be added to the lookahead plan LP in the previous step, we repair the current relaxed plan RP that have no applicable action by replacing one of its actions. Among the set of problem actions applicable in S, we heuristically choose an action. An effect of this action have to satisfy at least one precondition of one unsatisfied action of RP in S. The selected action is then added in LP, the replaced action is deleted from RP, and we return to the previous step. If no replacing action is found, we try to replace another action of RP.

The algorithm stops when no action can be added to the lookahead plan LP, or when all the actions of the relaxed plan RP are used.

Once again, the quality of the lookahead plan depends on the scheduling of actions in the relaxed plan. It also depends on the choice of the actions to be inserted in the lookahead plan. To improve this quality in the first step, the actions are inserted in the $failure$ list according to their difficulty of obtaining them (i.e the first layer in the relaxed graph where the action appears), and also by taking into account certain negative interactions between actions.

In the second step, the choice of an action in the set of applicable actions is done by taking into account the cost and the number of preconditions established by the action.

The classification, introduced by FF, of the applicable actions on the current state is also improved. Applicable actions on a current state S that belongs to the relaxed plan are said to be *helpful actions*. Applicable actions to S that belongs to the graph but not to the relaxed plan are said to be *rescue actions*. In opposition to the classification proposed in (Hoffmann & Nebel 2001), this partition keeps all the actions that can be applied to S and preserves the completeness and correctness of the algorithm. The best-first algorithm is modified to prefer the helpful actions over rescue actions.

Conclusion

In this paper, we present the computation of the heuristic implemented in rYAHSP starting from a numeric relaxed planning graph and its use with a lookahead strategy.

The preliminary results obtained by rYAHSP show that the concept of lookahead plan is interesting in the numeric framework. They also show that the choice of the actions and their scheduling in the relaxed plans and lookahead plans influence the metric of solution-plans. The combined use of various criteria can improve their quality.

References

Do, M. B., and Kambhampati, S. 2003. Sapa: A Multi-objective Metric Temporal Planner. *Journal of Artificial Intelligence Research* 20:155–194.

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.

Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *Proceedings of the 6th European Conference on Planning (ECP-01)*.

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

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

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *International Conference on Automated Planning and Scheduling*, 150–159.

The New Version of 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

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

Overview

CPT is a 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; Vidal & Geffner 2006). CPT competed in the optimal track of IPC-4, where it got a second place.

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 σ 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 σ, so that the partial plan σ 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 σ of the search tree. The novelty of CPT in relation to other temporal POCL planners such as IXTET (Laborie & Ghallab 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).

Currently, the semantics of the optimal temporal plans computed by CPT follows the one in (Smith & Weld 1999) where interfering actions (actions that delete a precondition or an effect of another one) 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). This restriction can in some domains produce slightly longer plans.

Additional pruning rules

CPT has been recently extended with several pruning rules. The primary goal of these rules was to give CPT the ability to solve planning problems in a suboptimal but backtrack-free way. Indeed, while fixing an upper bound on the makespan to a low value helps in pruning the search space, it has been remarked many times that fixing the bound to a high value (and thus, searching for a suboptimal plan) renders constraint-based planners particularly inefficient. To overcome this problem, we added many features to CPT, and we got interesting results in many classical benchmarks: BlocksWorld, Logistics, Gripper, Ferry, Satellite for example can be solved suboptimally without any backtrack. These results are reported in (Vidal & Geffner 2005). Some of these additions to CPT turn out to also help for optimal planning.

Impossible Supports

Many supports can be eliminated at preprocessing avoiding some dead-ends during the search. For example, the action $a' = putdown(b1)$ can never support the precondition $p = handempty$ of an action like $a = unstack(b1, b3)$. This is because action a has another precondition $p' = on(b1, b3)$

which is e-deleted[1] by a' (false after a') and which then would have to be reestablished by another action b before a. Yet it can be shown that in this domain, any such action b e-deletes p and is thus incompatible with the causal link $a'[p]a$.

More generally, let $dist(a', p, a)$ refer to a lower bound on the slack between actions a' and a in any valid plan in which *a' is a supporter of precondition p of a.* We show that for some cases, at preprocessing time, it can be shown that $dist(a', p, a) = \infty$, and hence, that a' can be safely removed from the domain of the variable $S(p, a)$ encoding the support of precondition p of a.

This actually happens when some precondition p' of a is not *reachable* from the initial situation that includes all the facts except those e-deleted by a' and where *the actions that either add or delete p are excluded.* The reason for this exclusion is that if a' supports the precondition p of a then it can be assumed that no action adding or deleting p can occur between a' and a (the first part is the systematicity requirement (McAllester & Rosenblitt 1991)). By a proposition being reachable we mean that it makes it into the so-called relaxed planning graph; the planning graph with the delete lists excluded (Hoffmann & Nebel 2001).

This simple test prunes the action $putdown(b_1)$ as a possible support of the precondition $handempty$ of action $unstack(b_1, b_3)$, the action $stack(b_1, b_3)$ as a possible support of precondition $clear(b_1)$ of $pickup(b_1)$, etc.

Unique Supports

We say that an action *consumes* an atom p when it requires and deletes p. For example, the actions $unstack(b_3, b_1)$ and $pickup(b_2)$ both consume the atom $handempty$. In such cases, if the actions make it into the plan, it can be shown that their common precondition p must have different supports. Indeed, if an action a deletes a precondition of a', and a' deletes a precondition of a, a and a' are incompatible and cannot overlap in time according to the semantics. Then either a must precede a' or a' must precede a, and in either case, the precondition p needs to be established at least twice: one for the first action, and one for the second. The constraint $S(p, a) \neq S(p, a')$ for pairs of actions a and a' that consume p, ensures this, and when one of the support variables $S(p, a)$ or $S(p, a')$ is instantiated to a value b, b is immediately removed from the domain of the other variable.

Distance Boosting

The distances $dist(a, a')$ precomputed for all pairs of actions a and a' provide a lower bound on the slack between the end of a and the beginning of a'. In some cases, this lower bound can be easily improved, leading to stronger inferences. For example, the distance between the actions $putdown(b_1)$ and $pickup(b_1)$ is 0, as it is actually possible to do one action after the other. Yet the action $putdown(b_1)$ followed by $pickup(b_1)$ makes sense only if some other action using the effects of the first, occurs between these two, as when block b_1 is on block b_2 but needs to be moved on top of the block beneath b_2.

Let us say that an action a *cancels* an action a' when 1) every atom added by a' is e-deleted by a, and 2) every atom added by a is a precondition of a'. Thus, when a cancels a', the sequence a', a does not add anything that was not already true before a'. For example, $pickup(b_1)$ cancels the action $putdown(b_1)$.

When an action a cancels a', and there is a precondition p of a that is made true by a' (i.e., p is added by a' and is mutex with some precondition of a'), the distance $dist(a', p, a)$ introduced above becomes ∞ if all the actions that use an effect of a' e-delete p. In such case, as before, the action a' can be excluded from the domain of the $S(p, a)$ variable. Otherwise, the distance $dist(a', a)$ can be increased to $\min_b[dist(a', b) + dist(b, a)]$ with b ranging over the actions different than a and a' that either use an effect of a' but do not e-delete p or do not use necessarily an effect of a' but add p (because a' may be followed by an action c before a that e-deletes p but only if there is another action b between c and a that re-establishes p).

In this way, the distance between the actions $putdown(a)$ and $pickup(a)$ in Blocks is increased by 2, the distance between $sail(a, b)$ and $sail(b, a)$ in Ferry is increased by 1, etc. The net effect is similar to pruning cycles of size two in standard heuristic search. Pruning cycles of larger sizes, however, appears to be more difficult in the POCL setting, although similar ideas can potentially be used for pruning certain sequences of commutative actions.

Improvement of the search algorithm

The original version of CPT performs a very basic backtracking search: its relative efficiency mainly comes from the look-ahead techniques encoded into the pruning rules, and from heuristics adapted to temporal planning. But even if an advanced look-ahead technique is used, one can be interested by looking for the reason of an encountered dead-end as finding the ideal ordering of variables is intractable in practice. A dead-end corresponds to a conflict between a subset of decisions (variable assignments) performed so far. In fact, it is relevant to prevent thrashing[2] to identify the most recent decision (let us call it the culprit one) that participates to the conflict. Indeed, once the culprit has been identified, we know that it is possible to safely backtrack up to it – this is the role of look-back techniques such as CBJ (Conflict-directed BackJumping) (Prosser 1993) and DBT (Dynamic Backtracking) (Ginsberg 1993).

A new approach as been recently proposed in (Lecoutre *et al.* 2006) to (indirectly) backtrack to the culprit of the last encountered dead-end. To achieve it, the leaf conflict variable becomes in priority the next variable to be selected as long as the successive assignments that involves it render

[1] An action a is said to *e-delete* an atom p when either a deletes p, a adds an atom q such that q and p are mutex, or a precondition r of a is mutex with p and a does not add p. In all cases, if a e-deletes p, p is false after doing a; see (Nguyen & Kambhampati 2001).

[2] Thrashing is the fact of repeatedly exploring the same subtrees. This phenomenon deserves to be carefully studied as an algorithm subject to thrashing can be very inefficient.

the network arc inconsistent. It then corresponds to checking the singleton consistency of this variable from the leaf towards the root of the search tree until a singleton value is found. In other words, the variable ordering heuristic is violated, until a backtrack to the culprit variable occurs and a singleton value is found. It is important to remark that, contrary to sophisticated backjump techniques, this technique can be grafted in a very simple way to a tree search algorithm without any additional data structure. This has been implemented very easily into CPT, making it able to solve difficult problems that were previously out of reach.

A note about the implementation

The first version of CPT planner was 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++. Due to a number of restrictions of this language, we made a completely new implementation using the C language. This implementation is based on a minimal Constraint Programming engine inspired by the Choco library, offering all the basic needs: CP variables with enumerated and bounded domains, automatic propagation on the change of the domains based on events (instantiation, removal, lower bound increased, ...), a complete backtrack mechanism for undoing the changes, and a basic backtracking algorithm. The constraints of CPT are implemented with propagation rules which are triggered by the underlying CP engine. This implementation is by far more efficient than the original one, also having very minimal memory requirements (except in some benchmark domains such as ZenoTravel, where the formulation of the domain by itself leads to a high number of variables with very large domains). This new version will soon be available on the CPT web page[3]. We also plan to release the minimal CP engine of CPT as a separate package, as it is completely independent of CPT and could serve as a basis for many different applications. As an example, CPT has been recently used in an evolutionary based approach of multi-objective temporal planning (Schoenauer, Savéant, & Vidal 2006).

Acknowledgments

Most of the work on CPT has been made with Héctor Geffner, as well as some material of this abstract borrowed from our common papers. Many thanks to him.

References

[Blum & Furst 1995] Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proceedings of IJCAI-95*, 1636–1642.

[Caseau, Josset, & Laburthe 1999] Caseau, Y.; Josset, F. X.; and Laburthe, F. 1999. CLAIRE: Combining sets, search and rules to better express algorithms. In *Proceedings of ICLP-99*, 245–259.

[Ginsberg 1993] Ginsberg, M. 1993. Dynamic backtracking. *Artificial Intelligence* 1:25–46.

[Haslum & Geffner 2001] Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *Proceedings of European Conference of Planning (ECP-01)*, 121–132.

[Hoffmann & Nebel 2001] Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 2001:253–302.

[Jonsson *et al.* 2000] Jonsson, A.; Morris, P.; Muscettola, N.; and Rajan, K. 2000. Planning in interplanetary space: Theory and practice. In *Proceedings of AIPS-2000*, 177–186.

[Joslin & Pollack 1996] Joslin, D., and Pollack, M. E. 1996. Is "early commitment" in plan generation ever a good idea? In *Proceedings of AAAI-96*, 1188–1193.

[Kautz & Selman 1999] Kautz, H., and Selman, B. 1999. Unifying SAT-based and Graph-based planning. In Dean, T., ed., *Proceedings of IJCAI-99*, 318–327. Morgan Kaufmann.

[Laborie & Ghallab 1995] Laborie, P., and Ghallab, M. 1995. Planning with sharable resources constraints. In Mellish, C., ed., *Proceedings of IJCAI-95*, 1643–1649. Morgan Kaufmann.

[Laburthe 2000] Laburthe, F. 2000. CHOCO: implementing a CP kernel. In *Proceedings of CP-00, Lecture Notes in CS, Vol 1894*. Springer.

[Lecoutre *et al.* 2006] Lecoutre, C.; Sais, L.; Tabary, S.; and Vidal, V. 2006. Last conflict based reasoning. In *Proceedings of ECAI-2006* (to appear).

[McAllester & Rosenblitt 1991] McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proceedings of AAAI-91*, 634–639. Anaheim, CA: AAAI Press.

[Nguyen & Kambhampati 2001] Nguyen, X. L., and Kambhampati, S. 2001. Reviving partial order planning. In *Proceedings of IJCAI-01*, 459–466.

[Prosser 1993] Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problems. *Computational Intelligence* 9(3):268–299.

[Schoenauer, Savéant, & Vidal 2006] Schoenauer, M.; Savéant, P.; and Vidal, V. 2006. Divide-and-evolve: a new memetic scheme for domain-independent temporal planning. In *Proceedings of EvoCOP-2006*, 247–260.

[Smith & Weld 1999] Smith, D., and Weld, D. S. 1999. Temporal planning with mutual exclusion reasoning. In *Proceedings of IJCAI-99*, 326–337.

[Van Beek & Chen 1999] Van Beek, P., and Chen, X. 1999. CPlan: a constraint programming approach to planning. In *Proceedings of AAAI-99*, 585–590.

[Vidal & Geffner 2004] Vidal, V., and Geffner, H. 2004. Branching and pruning: An optimal temporal POCL planner based on constraint programming. In *Proceedings of AAAI-2004*, 570–577.

[Vidal & Geffner 2005] Vidal, V., and Geffner, H. 2005. Solving simple planning problems with more inference and no search. In *Proceedings of CP-2005*, 682–696.

[Vidal & Geffner 2006] Vidal, V., and Geffner, H. 2006. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence* 170(3):298–335.

[Younes & Simmons 2003] Younes, H. L. S., and Simmons, R. G. 2003. VHPOP: Versatile heuristic partial order planner. *JAIR* 20:405–430.

[3]http://www.cril.univ-artois.fr/∼vidal/index.en.html

MaxPlan: Optimal Planning by Decomposed Satisfiability and Backward Reduction

Zhao Xing, Yixin Chen, and Weixiong Zhang
Department of Computer Science and Engineering
Washington University in St. Louis
Saint Louis, MO 63130
{zx2,chen,zhang}@cse.wustl.edu

Abstract

Planning as satisfiability is one of the best approaches to optimal planning. In addition to being effective and efficient on many planning domains, this approach is general, in that a generic method for satisfiability (SAT) can be used. As a result, it is able to take advantage of the general research devoted to SAT. Nevertheless, the potential of this approach has not been fully exploited. In the MaxPlan planner, we develop an efficient SAT solving algorithm that effectively exploits the structure of planning applications and decomposes the original SAT problem into a series of much simpler SAT subproblems. The decomposed SAT approach has been shown to be significantly more efficient than the previous approach which uses a generic SAT solver as a black-box without exploiting the problem structure. Several other novel techniques, including backward level reduction, accumulative learning of clauses, and search-space pruning based on multi-valued domain formulation, are also developed to further improve the search efficiency for both satisfiability solving and unsatisfiability proving.

1 Introduction

MaxPlan is an optimal planner for STRIPS-like propositional planning problems. It is optimal in terms of the number of parallel steps. It, in essence, follows the paradigm of planning as satisfiability [Selman & Kautz, 1992], which has emerged as one of the most effective formulations for optimal planning. The method of planning as satisfiability first transforms a STRIPS planning problem into a satisfiability (SAT) problem, and then solves the SAT problem using a generic SAT solver.

Representative planners under the paradigm of planning as satisfiability include Blackbox [Kautz & Selman, 1996] and SATPLAN [Kautz,]. The latest version of SATPLAN, SATPLAN04, has won the First Place prize in the optimal track of the Fourth International Planning Competition (IPC4). Despite its success, the current best realization of planning as satisfiability still has the following limitations.

a) SATPLAN performs a ***forward level expansion*** search that keeps increasing the estimated plan length and for each fixed length finds a solution or proves the problem unsolvable. We have noticed that most of the computing time that it spends on is to prove unsatisfiability, which could be expensive because the entire search space may need to be explored. This observation inspires us to search from the opposite direction, i.e. to reduce the estimated plan length from an upper bound. When the plan is long, however, finding a feasible solution can still be expensive. Effective strategies for reducing complexity are needed.

b) By transforming a planning problem into a SAT problem and solving the problem in a ”blackbox” fashion by a SAT solver, the structural and goal information in the planning problem is discarded. In contrast, heuristic search-based planning methods typically combine directed forward search with backward chaining to explicitly exploit information in the planning goals. This motivates us to study and utilize the internal structure of the SAT encoding.

c) The existing SAT planners solve a SAT problem as a whole, which becomes prohibitively expensive for large problems. One objective in the design of MaxPlan is to decompose a planning problem based on its structure in order to reduce search complexity.

d) To achieve the optimal time steps, the current realization of planning as satisfiability follows an incremental scheme in which the number of time steps is increased by one after each failed iteration. A careful examination of the process of the current approaches has revealed key insights into this general planning paradigm. First, the SAT problem instances derived at different time steps share similar structures. Second, the knowledge learnt from solving SAT instances for shorter time steps can be accumulated and used to speed up the processes for solving SAT instances of longer time steps. However, the existing SAT planners generate SAT problem instances independently from scratch and solve them in isolation so that knowledge learnt during previous iterations was lost.

The design of MaxPlan aims at overcoming the above limitations.

2 Overall Architecture of MaxPlan

Based on an action-based SAT formulation, MaxPlan employs a new decomposition scheme to decompose a planning problem into a series of SAT subproblems, one for each subgoal variable, in order to reduce the search complexity. Goal-oriented rules for selecting variables in SAT solvers are introduced to exploit logic structures of planning problems. Other novel techniques, including ***backward level reduction***, accumulative learning, and search space pruning based on a multi-valued formulation, are also developed and integrated with a generic SAT solver to further speed up the solution process.

The overall process of MaxPlan works as follows.

1. Estimate an upper bound of the optimal plan length. The search keeps tightening the upper bound until a plan length has been proved unsolvable.
2. For a given plan length, compile the input planning problem into a SAT formulation.
3. Solve the derived SAT problem using a novel SAT solver integrated with goal-oriented decomposition and search space pruning. Quit if the current plan depth is proven unsatisfiable.
4. After a plan depth is solved, use an accumulative learning scheme for taking advantage of the structural similarities among the SAT instances. Accumulate the knowledge (in the form of learnt clauses and pruning rules) to accelerate the processes of solving sebsequent instances.
5. Reduce the estimated optimal plan length and repeat from step 2.

MaxPlan significantly differs from the previous SAT-based planners in the overall direction of plan-length estimation and inter-level learning. Furthermore, instead of using a generic ”blackbox” SAT solver, MaxPlan integrates in the SAT solver a number of effective strategies for exploiting the structure of a planning problem.

3 Backward Level Reduction and Analysis of Action-Based Encoding

MaxPlan uses an approach to set the initial number of parallel steps which is different from that used by the previous SAT planners. Since the previous SAT planners use forward level expansion, they typically apply the Graphplan approach [Blum & Furst, 1997] to construct a relaxed planning graph to estimate a ***lower*** bound of the optimal parallel length. In contrast, since MaxPlan uses backward level reduction, it estimates an ***upper*** bound using a suboptimal planner to find a suboptimal sequential plan and parallelize it. We use the Fast Forward planner [Hoffmann & Nebel, 2001] in our current implementation.

To transform a planning problem into SAT, we use the same action-based encoding as used by SATPLAN04, which converts all fact variables to action variables.

We have closely studied the action-based SAT encoding. We classify the clauses into several classes, including the *E clauses* that are binary clauses generated from mutual exclusions, the *A clauses* that specifies the dependencies among actions, and the *G clauses* that specifies subgoals.

Most (but not all) variables in E clauses can be instantiated by applying unit propagation in a SAT solver. In other words, a majority of independent variables appear in the G clauses. As a result, the variables in the binary E clauses have a lower priority to be chosen. Therefore, the three classes of clauses should be treated differently. The G clauses should be treated as “core clauses” in SAT solving. In general, the G clauses only constitute a very small portion (typically less then 2 percents) of all the clauses. Focusing on branching those variables that appear in the G clauses first can significantly reduce the search space in SAT solving.

Most, if not all, existing SAT solvers lack mechanisms for exploiting structural information of real-world instances and for identifying independent variables. Identifying problem structural information is a hard problem. Being generic algorithms, these SAT solvers try to produce conflicts by branching on shorter clauses (such as heuristic [Li & Anbulagan, 1997] or the Jeroslow-Wang rule [Hooker & Vinay, 1995]), or attempt to detect conflicts by applying the locality of conflicts (such as the VSIDS heuristic [Moskewicz *et al.*, 2001]). These general heuristics can hardly detect structural properties of a planning problem.

4 Goal-Oriented Decomposition for SAT Solving

The majority part of the computation complexity of MaxPlan lies in solving the SAT problems at different parallel steps. In our implementation, we use a generic SAT solver, MiniSAT [E$\acute{e}$n & Biere, 2005], enhanced with our new strategies, as the SAT engine for MaxPlan. Note that the decomposition and space pruning strategies proposed in MaxPlan are general and can be incorporated into other SAT solvers.

As mentioned, SAT problems derived from real-world planning problems are often highly structured and contain a substantial portion of variables whose values can be determined by other variables through unit propagation. In other words, these variables are *dependent* variables. In contrast, those variables whose values cannot be directly inferred from the assignments of other variables are *independent* variables. Since the assignments of independent variables determine the values of most dependent variables, the number of assignments that need to be tried can be reduced significantly if we branch on independent variables first. In MaxPlan, independent variables are those appearing in the G clauses, and dependent variables are the ones not in the G clauses.

The complexity of SAT solving can be significantly reduced if we ignore the assignments of some independent variables. Therefore, in MaxPlan, we propose a ***goal-oriented decomposition*** scheme to decompose the SAT problem into a series of much simpler SAT subproblems with incremental involvement of the goal-oriented independent variables.

Suppose that there are N subgoals, we iteratively solve N subproblems. In solving the i^{th} subproblem, we set the variables for the first i subgoals true, and set free the other subgoal variables. After we solve the instance, based on the solution, we increase the priority scores of all action variables whose assignments are true by a constant amount in MiniSat in order to force to branch on these action variable and set them to true first in solving the next subproblem. Therefore, in each iteration, we focus on solving one subgoal. In this process, we also try to explore first the partial solution space that is close to the ones searched in the previous iterations by giving priorities to the related G clauses.

There are several advantages of the goal-oriented decomposition approach. Our preliminary experimental analysis has indicated that after the G variables have been fixed, the problem can be solved quickly by unit propagation. This means that the G variables are the most critically constrained variables in a planning problem. By incrementally increase the number of active critical G variables in a series of subproblems, the complexity of solving each subproblem is significantly reduced. As a result, the total complexity of solving all decomposed subproblems is still much smaller than the complexity of solving the original SAT problem without decomposition.

The proposed goal-oriented decomposition approach is different from incremental planning. A key difficulty of incremental planning is that it fixes the solutions of solved subproblems and often gets stuck at infeasible deadends. Our decomposition approach, on the contrary, allows backtracking and is a complete algorithm that can be used to prove unsatisfiability.

5 Search Space Pruning

In addition to goal-oriented decomposition, we develop two novel strategies to further reduce the search space of SAT solving. A common feature of the two strategies is that they both prune the search space by adding more constraints to the problem formulations.

5.1 Mutual Exclusion Constraints from Multi-Valued Formulations

In the SAT formulation, the E clauses encode binary mutual exclusion relations among actions. It is very difficult and expensive to detect all mutual exclusions, and most previous SAT planners only detect a small subsets of mutual exclusions. In MaxPlan, we use a new approach that can generate more mutual exclusions efficiently. The approach uses a multi-valued domain formulation (MDF) that translates the binary fact representation into a more compact representation with multi-valued variables. Each variable typically represents an invariant group for an object, and the variable can only take one value in the group at any time. For example, a truck may only be at one location at any time. The location of a truck is a variable, and different locations are the values that the variable may take.

Based on the MDF formulation, we can derive more mutual exclusions among actions. We first derive mutual exclusions among facts. In fact, all the binary facts associated with a multi-valued variable are mutually exclusive. For example, at(truck, location1) and at(truck, location2) are mutually exclusive. Based on the mutual exclusions among facts, we further derive mutual exclusions among actions based on the original definition of mutual exclusion [Blum & Furst, 1997]. The new mutual exclusions among actions are added to the action-based SAT formulation as additional constraints to prune infeasible regions of the search space. This technique improves the efficiency in both finding a feasible plan and proving unsatisfiability.

5.2 Accumulative Learning

An additional technique to further improve the search efficiency is accumulative learning. Typically a SAT solver can learn during search new redundant clauses which provide additional pruning power. This mechanism is called *clause learning*, and the derived clauses are called *learnt clauses*. In the existing SAT planners such as SATPLAN04, learnt clauses are not inherited across iterations. In each iteration, the SAT solver learns clauses from scratch.

Since MaxPlan uses an iterative process to search for an optimal plan, we note that a SAT instance at any iteration resembles the SAT instance for the previous iteration, except for some G clauses due to the change of plan length. We take advantage of such a structural similarity and propose the accumulative learning scheme. This scheme has two key components. Instead of re-encoding the whole problem from scratch after each iteration, we simply modify and patch the previous encoding to specify the new constraints of the next iteration. Therefore, the time for encoding can be significantly reduced, and such saving can be significant for large planning problems. More importantly, we can retain all the learnt clauses (not related to goal clauses) in all the previous iterations and re-use them in the next iteration. As a result, most learnt clauses only need to be learnt once, which saves running time.

It is important to note that the learnt clauses that are retained for the next iteration may be learnt again in the next iteration if they were not retained. Most existing efficient SAT solvers have efficient mechanisms for clause learning that support managing and deleting learnt clauses intelligently. Therefore, our accumulative learning scheme incurs very little additional memory over the original SAT solver.

6 Conclusions

In summary, MaxPlan follows the general paradigm of planning as satisfiability and incorporates significant extensions. First, MaxPlan uses backward level reduction instead of the previous forward expansion approach. Second, MaxPlan uses a novel goal-oriented decomposition method, developed in this paper, to greatly improve the efficiency of solving SAT problems derived from planning. This decomposition method exploits structures of a planning problem and the effects of different classes of clauses in the SAT encoding. Finally, two new strategies, MDF-based constraints and accumulative learning, are developed to further prune the search space.

References

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

[Eén & Biere, 2005] Eén, N., and Biere, A. 2005. Effective preprocessing in SAT through variable and clause elimination. In *SAT*.

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

[Hooker & Vinay, 1995] Hooker, J., and Vinay, V. 1995. Branching rules for satisfiability. *J. Automated Reasoning* 15:359–383.

[Kautz & Selman, 1996] Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of AAAI-96*, 1194–1201.

[Kautz,] Kautz, H. SATPLAN04: Planning as satisfiability. IPC4 abstract, 2004.

[Li & Anbulagan, 1997] Li, C., and Anbulagan. 1997. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of IJCAI-97*, 366–371.

[Moskewicz *et al.*, 2001] Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*.

[Selman & Kautz, 1992] Selman, B., and Kautz, H. 1992. Planning as satisfiability. In *Proceedings ECAI-92*, 359–363.

Abstracting Planning Problems with Preferences and Soft Goals

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

Abstract

In this paper we describe the planner PATTERNPLAN that participates in the *Fifth International Planning Competition*. We put emphasis on solving the *over-subscription planning* problem, which characterized the major difference between this planning competition and the previous ones. Our solution is to abstract the over-subscription problem into a generalized *orienteering problem* and use the solution of this problem to guide the search of solving the original over-subscription problem.

Introduction

The goal in traditional planning research is usually to generate a plan that satisfies a fixed set of subgoals, sometimes with the additional objective of minimizing costs. This also has been the theme in previous planning competitions(McDermott 2000; Bacchus 2001; Long & Fox 2003; Hoffmann & Edelkamp 2005). In many real-world planning problems, however, it is necessary to explicitly consider the trade-off between the rewards of achieving goals against their costs. Very recently this problem was considered in the setting of *over-subscription planning* (Smith 2004). In this setting, there are a large number of possible goals with differing rewards. Given a limit of time and resources, the planning system must choose to accomplish a subset of those goals that maximizes the rewards. In the *Fifth International Planning Competition* (Gerevini & Long 2005), this problem is considered more generally by setting the objective as optimizing a linear combination of rewards and costs. We also call these problems over-subscription planning problems.

Given an over-subscription planning problem, there is a cost and a reward associated with each valid plan. For any over-subscription problem, there is a underlying relation between rewards and costs. The over-subscription problem is implicitly a linear optimization problem on this relation. Because exactly generating this relation is difficult, previous approaches employed various approximation methods (Smith 2004; van den Briel *et al.* 2004; Nigenda & Kambhampati 2005; Benton, Do, & Kambhampati 2005).

We also solve the over-subscription problem approximately by generating an abstraction problem. By design the reward-cost relation on the abstraction problem approximates that on the original problem. A solution with good quality on the abstraction problem thus hopefully leads to a solution with good quality on the original problem. With such a guide, we employ a forward-chaining method to solve the original over-subscription problem.

In the following section, we will describe the abstraction method and the planning method in detail.

Abstraction

We take an approach that is similar in spirit with a previous method that constructs *orienteering problems* (Keller 1989; Blum *et al.* 2003) for over-subscription planning (Smith 2004). Our approach differs in two ways. First, we recognize that a generalization of the orienteering problem is necessary. Second, we introduce a novel fully automatic method that translates the over-subscription problem into a generalized orienteering problem.

We start by translating the PDDL planning problems into the SAS^+ formalism (Bäckström & Nebel 1995), based on methods previously published (Helmert 2004; Edelkamp & Helmert 1999). In SAS^+ variables can have non-binary finite domains. More compact representations are achieved with SAS^+ than with STRIPS or PDDL.

Starting with an empty abstract problem, we then incrementally introduce variables to the abstract problem until a pre-defined threshold (e.g., a certain number of variables) is reached. Only those variables that are both a precondition and an effect of *some* action are included. For each such action, the cost of that action is associated with the variable. The variables are added to the abstract problem in descending order of their minimal associated costs. This abstract state space represents the part of the original problem that is most sensitive to costs. Any action or any set of goals can be projected onto this abstract state space by ignoring other variables.

In the next step all the goals with rewards are *embedded* into this abstract state space. A goal can be embedded into an abstract state if and only if the abstract state satisfies the goal after the goal is projected into the abstract state space. Note that some abstract state might contain no goal. Also note it is possible that the same goal is embedded into multiple abstract states. Because of this we have to introduce a generalization of the orienteering problem.

Now we are ready to define the generalized orienteering problem. This problem consists of the abstract state space, the distance between any pair of states, the goals that are associated with states, and a reward for each goal. Starting from the projection state of the initial original state, the objective is to optimize a linear combination of the rewards and costs by moving on the abstract state space. A reward for each goal can be collected only once, even if the same goal is available in multiple abstract states.

We solve the generalized orienteering problem by adapting heuristic local search methods that are used to solve the original orienteering problems (Smith 2004).

The solution for the generalized orienteering problem defines an ordering on the goals of the original over-subscription problem. We then use a black-box cost-optimizing planner to solve enlarging subsets of the goals in sequence. The subset goal initially contains the first goal solved in the generalized orienteering problem. At each stage the planner tries to achieve all the goals in the subset goal. A new goal is added to the subset goal if the planner succeeds. The black-box planner is stopped whenever a subset goal is not achievable within a time limit. We call the plan that successfully achieved the second last subset goal a *preliminary plan*. The *final plan* is a prefix of the preliminary plan that achieves the linear optimization of costs and rewards.

References

Bacchus, F. 2001. The AIPS ’00 planning competition. *AI Magazine* 22(3):47–56.

Bäckström, C., and Nebel, B. 1995. Complexity results for sas+ planning. *Computational Intelligence* 11:625–656.

Benton, J.; Do, M. B.; and Kambhampati, S. 2005. Over-subscription planning with numeric goals. In *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)*, 1207–1213.

Blum, A.; Chawla, S.; Karger, D. R.; Lane, T.; Meyerson, A.; and Minkoff, M. 2003. Approximation algorithms for orienteering and discounted-reward TSP. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, 46–55.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proceedings of the European Conference on Planning (ECP)*, 135–147.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences for PDDL3. Technical Report R.T. 2005-08-07, Department of Electronics for Automation, University of Brescia, Brescia, Italy.

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

Hoffmann, J., and Edelkamp, S. 2005. The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research (JAIR)* 24:519–579.

Keller, C. P. 1989. Algorithms to solve the orienteering problem: A comparison. *European Journal of Operational Research* 41(2):224–231. No online version.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research (JAIR)* 20:1–59.

McDermott, D. V. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–55.

Nigenda, R. S., and Kambhampati, S. 2005. Planning graph heuristics for selecting objectives in over-subscription planning problems. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 192–201.

Smith, D. E. 2004. Choosing objectives in over-subscription planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 393–401.

van den Briel, M.; Nigenda, R. S.; Do, M. B.; and Kambhampati, S. 2004. Effective approaches for partial satisfaction (over-subscription) planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 562–569.

POND: The Partially-Observable and Non-Deterministic Planner

Daniel Bryce
Department of Computer Science and Engineering
Arizona State University, Brickyard Suite 501
699 South Mill Avenue, Tempe, AZ 85281
dan.bryce@asu.edu

Abstract

This paper describes *POND*, a planner developed to solve problems characterized by partial observability and non-determinism. *POND* searches in the space of belief states, guided by a relaxed plan heuristic. Many of the more interesting theoretical issues showcased by *POND* show up within its relaxed plan heuristics. Namely, the exciting topics are defining distance estimates between belief states, efficiently computing such distance estimates on planning graphs, and sharing planning graphs and relaxed plans between belief states.

Introduction

The *POND* planner solves many types of planning problems characterized by uncertainty, whether they are non-deterministic/probabilistic, are non/partially observable, or have deterministic/uncertain actions. *POND* accepts PPDDL-like[1] (Younes & Littman 2004) problem descriptions and generates conformant and conditional plans. *POND* searches forward in the space of belief states, similar to GPT (Bonet & Geffner 2000), using various search algorithms (A*, AO*, LAO*, Enforced Hill-Climbing) depending on the problem and user preferences. To compute heuristics for search, *POND* can use several different planning graph techniques. We start by discussing some of the theory that goes into computing the planning graph heuristics used by *POND*, and then describe the planner implementation.

Theory

Since *POND* can handle several types of planning problems, we concentrate on the techniques used for conformant non-deterministic planning. We refer to (Bryce, Kambhampati, & Smith 2006a) and (Bryce, Kambhampati, & Smith 2006b) for additional techniques, not described here.

[1]PPDDL extended for various things such as non-determinism, observations, goal probability thresholds, etc.

Belief State Distance: To search in belief space, *POND* estimates the conformant plan distance between the belief state(s) at the end of its current plan prefix and a goal belief state. The distance between belief states is taken as an aggregate measure of the underlying distances between states in the belief states. For instance, a possible admissible measure would find the minimum distance from every state in the current belief state to a state in the goal belief state, then take the maximum of these (this is the measure used by GPT). Since taking the maximum of the minimum state distances assumes full positive interaction between the states, we would not account for many of the actions that differ between the sequences for each state (i.e., miss independence). Taking the summation of the minimum state distances would assume full independence, but miss the positive interaction. Instead, we use a measure that exploits both positive interaction and independence. By analogy to plan merging, we would like to merge the action sequences for each of the states in the current belief state so that actions *overlap* as much as possible (Bryce, Kambhampati, & Smith 2006a). The resulting merged plan contains all actions used in common or independently by the different states in the belief state. We can obtain this measure by computing a classical relaxed plan for each state in our current belief state and merging the relaxed plans. However, there may be many states in our belief state and computing a planning graph for each state is costly.

Heuristic Computation: In order to compute our belief state distance measure without constructing multiple explicit planning graphs, we use a planning graph generalization, called the Labeled Uncertainty Graph (*LUG*) (Bryce, Kambhampati, & Smith 2006a). The *LUG* represents multiple explicit planning graphs implicitly. The idea is to use a single planning graph skeleton to represent common action and proposition connectivity, and use annotations (labels) that denote which planning graph components exist in the explicit planning graphs. Labels are propositional formulas, whose models correspond to states in the belief state. We can determine which explicit planning graphs contain a proposition by examining the models of the proposition's label. If a state entails the label of the proposition at level k, the proposition is in the explicit planning graph for the state at level k.

Using LUG connectivity, we can determine which actions are needed to support the goal propositions, and using labels we know when we have chosen enough actions to support the goals from each of the states in our belief state. Thus, we can extract a relaxed plan that represents an implicitly merged plan for each of the states in our belief state. This relaxed plan indicates the plan distance to transition each of the states in a belief state to a goal belief state.

State Agnostic Planning Graphs: The LUG implicitly represents a set of explicit planning graphs. Using a state agnostic planning graph (SAG) (Cushing & Bryce 2005), a generalization of the LUG, we can build a LUG for every possible state. The SAG and LUG are identical except for which states are represented and how we compute relaxed plans. To extract the relaxed plan for a belief state from the SAG (assuming the belief state is represented by a propositional formula) we need to take the conjunction of each label with the belief state to reveal the LUG for the belief state. Those planning graph elements where the conjunction is satisfiable are in the revealed LUG. By computing the SAG, we construct a single, sometimes costly, LUG whose cost is amortized over each belief state.

Global Relaxed Plan: Alternative to using the SAG to compute a relaxed plan for each belief state, we can compute a global relaxed plan. The global relaxed plan continues the SAG generalization by making a state agnostic relaxed plan. We extract the global relaxed plan, which is a relaxed plan for the belief state containing all states, and then for each belief state encountered in search we restrict the global relaxed plan to the actions needed for the belief state. By restricting the global relaxed plan, we mean that we take the conjunction of each action's label in the global relaxed plan with the belief state formula. Those actions where the conjunction is satisfiable are in the relaxed plan. The global relaxed plan is admittedly less accurate than extracting a relaxed plan from the SAG for a specific belief state, but is very fast to compute.

Lazily Enforced Hill-Climbing: $POND$ uses a lazily enforced hill-climbing search, guided by three heuristics: belief state cardinality (Bertoli, Cimatti, & Roveri 2001), the global relaxed plan and the SAG relaxed plan. These heuristics have an increasing computation cost (and accuracy). The basic idea is to the use the cardinality heuristic in hill-climbing, as long as it improves the heuristic distance to the goal belief state. If the cardinality of the current children search nodes does not decrease, then we re-evaluate the children with the global relaxed plan (a slightly more costly, but better heuristic). If the global relaxed plan cannot find a better child, then we switch to the SAG relaxed plan in an A* search rooted at our current search node.

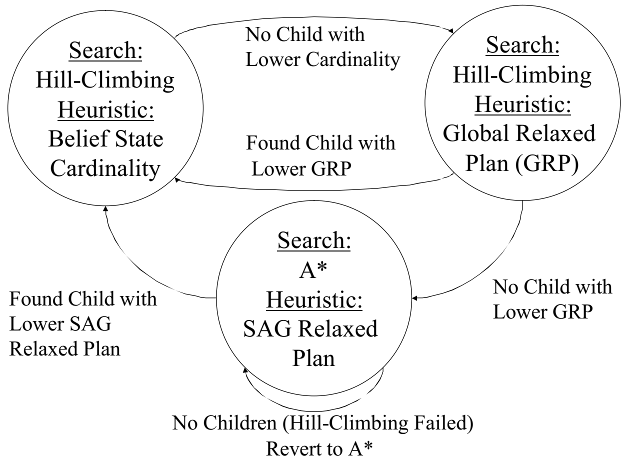
Search:
Hill-Climbing
Heuristic:
Belief State Cardinality
No Child with Lower Cardinality
Search:
Hill-Climbing
Heuristic:
Global Relaxed Plan (GRP)
Found Child with Lower GRP
Search:
A*
Heuristic:
SAG Relaxed Plan
Found Child with Lower SAG Relaxed Plan
No Child with Lower GRP
No Children (Hill-Climbing Failed) Revert to A*

Figure 1: Lazily Enforced Hill-Climbing strategy.

The SAG relaxed plan is the costliest heuristic, but the most informed. Once the A* search finds a better cost child, we resume hill-climbing with cardinality. If search fails, we revert to A* search with the global relaxed plan heuristic. The automata in Figure 1 depicts our search strategy. Since the heuristic landscape defined by cardinality, the global relaxed plan, and SAG relaxed plan are different, we say the search is lazily enforced hill-climbing. It is not always the case that using one heuristic to escape a heuristic plateau of another heuristic decreases the original distance to the goal belief state. However, since we use the heuristics in order of increasing accuracy, we are more confident in the direction chosen by the heuristics even if it means an increase in the original heuristic distance.

Implementation

$POND$ is implemented in C++ and uses several existing technologies. It employs the PPDDL parser (Younes & Littman 2004) for input, the IPP planning graph construction code (Koehler *et al.* 1997) for the LUG, and the CUDD BDD package (Somenzi 1998) for representing belief states, actions, and labels. $POND$ resembles MBP (Bertoli *et al.* 2001) because it uses BDDs to represent belief states and actions, and uses BDD operations to symbolically compute the transition between belief states. $POND$ is perhaps most similar to KACMBP (Bertoli & Cimatti 2002) because we use both cardinality and reachability heuristics, however our reachability heuristics are based on conformant relaxed plans.

Acknowledgements: This work was supported by the NSF grant IIS-0308139, the ONR grant N000140610058, the MCT/NASA summer fellowship program, the ARCS foundation, an IBM faculty award, and Arizona State University. We would like to thank Subbarao Kambhampati, David E. Smith, and William Cushing for contributing to the theory used in $POND$.

Much additional thanks is also given to William Cushing for help with implementation.

References

Bertoli, P., and Cimatti, A. 2002. Improving heuristics for planning as search in belief space. In *Proceedings of AIPS'02*.

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

Bertoli, P.; Cimatti, A.; and Roveri, M. 2001. Heuristic search + symbolic model checking = efficient conformant planning. In *Proceedings of IJCAI'01*.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *Proceedings of ECP'99*.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Proceedings of AIPS'00*.

Bryce, D.; Kambhampati, S.; and Smith, D. 2006a. Planning graph heuristics for belief space search. *JAIR*. (To appear).

Bryce, D.; Kambhampati, S.; and Smith, D. 2006b. Sequential monte carlo in probabilistic planning reachability heuristics. In *Proceedings of ICAPS'06*.

Cushing, W., and Bryce, D. 2005. State agnostic planning graphs. In *Proceedings of AAAI'05*.

Koehler, J.; Nebel, B.; Hoffmann, J.; and Dimopoulos, Y. 1997. Extending planning graphs to an adl subset. In *Proceedings of ECP'97*.

Somenzi, F. 1998. *CUDD: CU Decision Diagram Package Release 2.3.0*. University of Colorado at Boulder.

Younes, H., and Littman, M. 2004. PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. Technical report, CMU-CS-04-167, Carnegie Mellon University.

Conformant-FF

Jörg Hoffmann
Max Planck Institute for CS
Saarbrücken, Germany
hoffmann@mpi-sb.mpg.de

The Conformant-FF planner, as entered into (the conformant track of) IPC-5, is exactly the system as described by (Brafman & Hoffmann 2004; Hoffmann & Brafman 2006). The only differences amount to a slightly changed input syntax, as used in IPC-5. The planner is an extension of the FF system (Hoffmann & Nebel 2001). The main trick in the extension is the use of SAT-based techniques to reason about uncertainty.

Conformant-FF, in its current implementation, can deal with initial state sets given as CNF formulas, actions with conditional effects, full ADL in effect conditions, and a subset of ADL in action preconditions and the goal formula. Constructs not supported are disjunction in preconditions and the goal (unless the disjunction disappears when processing static facts), and, most importantly, nondeterministic action effects.

Conformant-FF performs a search in belief space, as suggested first with the GPT system (Bonet & Geffner 2000), and also done in the various versions of the MBP system (Cimatti & Roveri 2000; Bertoli & Cimatti 2002). A belief state is the set of world states that are possible at some time point. The belief space is the space of all belief states reachable from the initial belief state. There are two key differences between the three systems: (1) their *representation* of belief states; (2) the *heuristic* used to guide the search.

GPT represents the belief states explicitly, enumerating the respective world states. Standard heuristic functions (from deterministic planning) can then be aggregated appropriately. MBP represents the belief states symbolically, i.e., each belief state is now a BDD. The heuristic simply prefers belief states with less uncertainty, i.e., BDDs that represent a smaller state set. Conformant-FF uses a very lazy representation of belief states, including only a *partial* knowledge: for a belief state B, it just computes the facts $T(B)$ that are true in *all* world states $s \in B$. This knowledge suffices to do STRIPS-style conformant planning: an action precondition *pre* (a conjunction of facts) is satisfied in B iff $pre \subseteq T(B)$; the goal G (a conjunction of facts) is satisfied in B iff $G \subseteq T(B)$. The facts $T(B)$ are computed by encoding the semantics of the action sequence leading to B as a “time-stamped” CNF formula ϕ, defining how fact values change over the action sequence, in a straightforward way. Conjoining ϕ with the initial state formula ϕ^I, one gets that $p \in T(B)$ iff $\phi^I \wedge \phi \models p$.

Conformant-FF’s lazy or “implicit” representation can be seen as a way of trading space for time: on the positive side, we do not need to keep full detail about each B in memory; on the negative side, not having full detail about B forces us to reason all the way back to the initial state (in building the formulas ϕ) when computing the successors to B. In practice, we found even naive SAT solvers to be extremely efficient in solving the formulas arising in this context, so that the runtime price to pay is, in most cases, low.

The probably more crucial novelty in Conformant-FF is its heuristic function. This is an extension of FF’s “relaxed plan heuristic” to the conformant setting, i.e., to initial states given as CNF formulas. The underlying relaxation is still to ignore the delete lists. Relaxed planning, however, is still **co-NP**-hard when the initial “state” is a CNF. We get around this by making another relaxation: we ignore all but one of the (unknown) effect conditions of each effect. This corresponds to a 2-projection of the CNF formula that would encode the semantics of the relaxed actions. To obtain a polynomial worst-case behavior, one would also have to 2-project the initial state formula. We tried this, and found it to produce very bad heuristic values in many examples. So, instead, we keep the initial state formula unchanged, investing the effort to reason about it with a SAT solver. This produces good heuristic values in many cases, with a tolerable overhead since the initial state formula is typically neither overly large nor overly complicated.

In the traditional conformant benchmarks (Bombs, Ring, Cube, ...) Conformant-FF is sometimes competitive with GPT and MBP, sometimes outperformed vastly (particularly in Ring). In benchmarks created as classical benchmarks enhanced with uncertainty, however, Conformant-FF is many orders of magnitude superior to both GPT and (all variants of) MBP, since the heuristic function inherits, to a large extent, the quality of FF’s heuristic function in the classical setting.

References

Bertoli, P., and Cimatti, A. 2002. Improving heuristics for planning as search in belief space. In Ghallab, M.; Hertzberg, J.; and Traverso, P., eds., *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02)*, 143–152. Toulouse, France: Morgan Kaufmann.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In Chien, S.; Kambhampati, R.; and Knoblock, C., eds., *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS-00)*, 52–61. Breckenridge, CO: AAAI Press, Menlo Park.

Brafman, R., and Hoffmann, J. 2004. Conformant planning via heuristic forward search: A new approach. In Koenig, S.; Zilberstein, S.; and Koehler, J., eds., *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, 355–364. Whistler, Canada: Morgan Kaufmann.

Cimatti, A., and Roveri, M. 2000. Conformant planning via symbolic model checking. *JAIR* 13:305–338.

Hoffmann, J., and Brafman, R. 2006. Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence* 170(6–7):507–541.

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

COMPLAN: A Conformant Probabilistic Planner[*]

Jinbo Huang
Logic and Computation Program
National ICT Australia
Canberra, ACT 0200 Australia
jinbo.huang@nicta.com.au

Abstract

COMPLAN is a conformant probabilistic planner that finds a plan with maximum probability of success for a given horizon. The core of the planner is a a depth-first branch-and-bound search in the plan space. For each potential search node, an upper bound is computed on the success probability of the best plans under the node, and the node is pruned if this upper bound is not greater than the success probability of the best plan already found. A major source of efficiency for this algorithm is the efficient computation of these upper bounds, which is possible by encoding the original planning problem as a propositional formula and compiling the formula into deterministic decomposable negation normal form.

Conformant Probabilistic Planning

Consider the SLIPPERY-GRIPPER domain (Kushmerick, Hanks, & Weld 1995), where a robot needs to have a block painted and held in his gripper, while keeping his gripper clean. The gripper starts out clean, but may be wet, which may prevent him from holding the block; painting the block, while certain to succeed, may make his gripper dirty; a dryer is available to dry the gripper. Given probability distributions quantifying these uncertainties and a planning horizon, the robot requires a plan that achieves the goal with maximum probability.

A probabilistic planning problem can be characterized by a tuple $\langle \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{G}, n\rangle$ where $\mathbf{S}$ is the set of possible world states, $\mathbf{I}$ is a probability distribution over $\mathbf{S}$ quantifying the uncertainty about the initial state, $\mathbf{A}$ is the set of actions, all of which are assumed to be applicable in every state, $\mathbf{G}$ is the set of goal states, and n is the planning horizon.

To quantify the uncertainty in action effects, each action $a \in \mathbf{A}$ is a function that maps each state $s \in \mathbf{S}$ to a probability distribution Pr_s^a over all states $\mathbf{S}$. Starting from our initial *belief state* B_0, which is equal to $\mathbf{I}$, each action $a \in \mathbf{A}$ taken will bring us to a new belief state with updated probabilities for the world states $\mathbf{S}$:

$$B_n(s') = \sum_{s \in \mathbf{S}} B_{n-1}(s) \cdot Pr_s^a(s').$$

A solution to the *conformant* probabilistic planning problem is then a sequence of n actions, or an n-step plan, leading to belief state B_n, such that the sum of the probabilities assigned by B_n to the goal states is maximized.

Propositional Encoding

A probabilistic planning problem can be encoded by a propositional formula, where a subset of the propositional variables are labeled with probabilities (Littman 1997). As an example we consider the encoding of SLIPPERY-GRIPPER. The state space $\mathbf{S}$ of SLIPPERY-GRIPPER can be encoded by four propositional variables: BP (block-painted), BH (block-held), GC (gripper-clean), GD (gripper-dry). Suppose initially the block is not painted and not held, and the gripper is clean but dry only with probability 0.7.

To encode this initial belief state, we introduce a *chance* variable p, and label it with the number 0.7. We then write:

$$\neg\mathsf{BP}, \neg\mathsf{BH}, \mathsf{GC}, \neg p \vee \mathsf{GD}, p \vee \neg\mathsf{GD}.$$

This five-clause formula has the property that each setting of variable p will simplify the formula, resulting in a single world state, whose probability is given by the label of p (if p is set to $true$), or 1 minus that (if p is set to $false$).

We now consider the encoding of uncertain action effects. First we introduce a new set of state variables—$\mathsf{BP}', \mathsf{BH}', \mathsf{GD}', \mathsf{GC}'$—to encode the states reached after executing an action. There are three actions available: $\mathbf{A} = \{\mathsf{dry}, \mathsf{paint}, \mathsf{pickup}\}$. Suppose action dry dries a wet gripper with probability 0.8, and does not affect a dry gripper. We introduce a chance variable q, label it with 0.8, and write:

$$\neg\mathsf{dry} \vee \mathsf{GD} \vee \neg q \vee \mathsf{GD}', \neg\mathsf{dry} \vee \mathsf{GD} \vee q \vee \neg\mathsf{GD}', \neg\mathsf{dry} \vee \neg\mathsf{GD} \vee \mathsf{GD}'.$$

We also need a set of clauses, known as a *frame axiom*, saying that the other variables are not affected by the action:

$$\neg\mathsf{dry} \vee (\mathsf{BP} \Leftrightarrow \mathsf{BP}'), \neg\mathsf{dry} \vee (\mathsf{BH} \Leftrightarrow \mathsf{BH}'), \neg\mathsf{dry} \vee (\mathsf{GC} \Leftrightarrow \mathsf{GC}').$$

After all actions are encoded, we write the following saying that exactly one of the three actions will be taken:

$$\mathsf{dry} \vee \mathsf{paint} \vee \mathsf{pickup},$$

$$\neg\mathsf{dry} \vee \neg\mathsf{paint}, \neg\mathsf{dry} \vee \neg\mathsf{pickup}, \neg\mathsf{paint} \vee \neg\mathsf{pickup}.$$

Finally, our goal that the block be painted and held and the gripper be clean translates into three unit clauses:

$$\mathsf{BP}', \mathsf{BH}', \mathsf{GC}'.$$

This completes our propositional encoding of the planning problem, for horizon 1. The resulting set of clauses Δ_1 can be characterized as the conjunction of three components:

$$\Delta_1 \equiv \mathcal{I}(P_{-1}, S_0) \wedge \mathcal{A}(S_0, A_0, P_0, S_1) \wedge \mathcal{G}(S_1),$$

[*]This document is based on (Huang 2006).

where $\mathcal{I}(P_{-1}, S_0)$ is a set of clauses over the initial chance variables P_{-1}, and the state variables S_0 at time 0, encoding the initial belief state; $\mathcal{A}(S_0, A_0, P_0, S_1)$ is a set of clauses over the state variables S_0, action variables A_0, and chance variables P_0 at time 0, and state variables S_1 at time 1, encoding the action effects; and $\mathcal{G}(S_1)$ is a set of clauses over the state variables S_1 at time 1, encoding the goal condition.

From this characterization, an encoding Δ_n for n-step planning can be produced in a mechanical way by repeating the middle component with a new set of variables for each additional time step, and updating the goal condition:

$$\Delta_n \equiv \mathcal{I}(P_{-1}, S_0) \wedge \mathcal{A}(S_0, A_0, P_0, S_1) \wedge \mathcal{A}(S_1, A_1, P_1, S_2) \wedge \ldots \wedge \mathcal{A}(S_{n-1}, A_{n-1}, P_{n-1}, S_n) \wedge \mathcal{G}(S_n). \quad (1)$$

In this encoding, an n-step plan is an instantiation π of the action variables $A = A_0 \cup A_1 \cup \ldots \cup A_{n-1}$, an *eventuality* is an instantiation of the chance variables $P = P_{-1} \cup P_0 \cup \ldots \cup P_{n-1}$, and the probability of an eventuality is given by multiplying together the label of each chance variable, or 1 minus that, depending on its sign in the instantiation. A solution to the conformant probabilistic planning problem is then a plan π^* such that the sum of the probabilities $Pr(\epsilon)$ of all eventualities ϵ consistent with π is maximized:

$$\pi^* = \arg\max_{\pi} \sum_{\pi \wedge \epsilon \wedge \Delta_n \text{is consistent}} Pr(\epsilon). \quad (2)$$

Compilation to Deterministic DNNF

COMPLAN exploits the particular structure of probabilistic planning problems, as characterized by Equation 1, by compiling Δ_n into deterministic decomposable negation normal form (deterministic DNNF, or d-DNNF) (Darwiche & Marquis 2002) using the publicly available C2D compiler (Darwiche 2004; 2005), before the search starts.

Deterministic DNNF A propositional formula is in d-DNNF if it (i) only uses conjunction, disjunction, and negation, and negation only appears next to variables; and (ii) satisfies *decomposability* and *determinism*. Decomposability requires that conjuncts of any conjunction share no variables; determinism requires that disjuncts of any disjunction be pairwise inconsistent. The formula shown in Figure 1, for example, is in d-DNNF, and is equivalent to the 2-step encoding Δ_2 of SLIPPERY-GRIPPER, after the state variables $S = S_0 \cup S_1 \cup S_2$ have been existentially quantified.

Recall that existential quantification of a variable is defined as: $\exists x.\Delta \equiv \Delta|_x \vee \Delta|_{\overline{x}}$, where $\Delta|_x$ ($\Delta|_{\overline{x}}$) denotes setting variable x to $true$ ($false$) in Δ. In these planning problems, existential quantification of the state variables is useful because these variables do not appear in Equation 2 and their absence can reduce the size of the problem.

Efficient Plan Assessment Compilation of the planning problem into d-DNNF provides an efficient method for plan assessment: The computation of the success probability of any complete plan π for the n-step planning problem Δ_n,

$$Pr(\pi) = \sum_{\pi \wedge \epsilon \wedge \Delta_n \text{is consistent}} Pr(\epsilon), \quad (3)$$

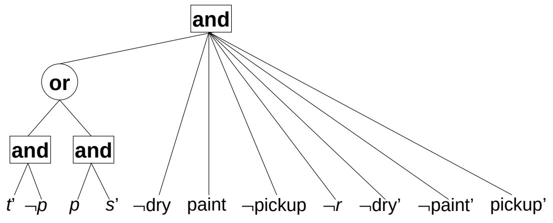

Figure 1: The 2-step SLIPPERY-GRIPPER in d-DNNF.

which is necessary for computing Equation 2, can be done in time linear in the size of a d-DNNF compilation of $\exists S.\Delta_n$, where S is the set of the state variables.

The source of this efficiency lies in that the d-DNNF graph can be viewed as a *factorization* of Equation 3, by regarding the conjunction nodes as multiplications and disjunction nodes as summations. Specifically, given the label $Pr(p)$ of each chance variable p, and a plan π, $Pr(\pi)$ can be obtained by a single bottom-up traversal of the d-DNNF graph, where a value is assigned to each node N of the graph as described in Algorithm 1.

Algorithm 1 Plan Assessment

$$val(N, \pi) = \begin{cases} 1, & \text{if } N \text{ is a leaf node that mentions an action variable and is consistent with } \pi; \\ 0, & \text{if } N \text{ is a leaf node that mentions an action variable and is inconsistent with } \pi; \\ Pr(p), & \text{if } N \text{ is a leaf node } p \text{ where } p \text{ is a chance variable;} \\ 1 - Pr(p), & \text{if } N \text{ is a leaf node } \neg p \text{ where } p \text{ is a chance variable;} \\ \prod_i val(N_i, \pi), & \text{if } N = \bigwedge_i N_i; \\ \sum_i val(N_i, \pi), & \text{if } N = \bigvee_i N_i. \end{cases}$$

The value assigned to the root is then $Pr(\pi)$. For example, given the 2-step plan paint-pickup′ for SLIPPERY-GRIPPER, which is actually a 6-variable instantiation (¬dry, paint, ¬pickup, ¬dry′, ¬paint′, pickup′), Algorithm 1 run on Figure 1 computes its success probability as 0.7335.

Depth-First Branch-and-Bound

Given a method for plan assessment (Algorithm 1), an optimal conformant plan can be found by a systematic search in the space of all possible plans. COMPLAN uses a depth-first branch-and-bound for this purpose, where upper bounds on values of partial plans are efficiently computed by traversing the d-DNNF compilation of the planning problem.

Computing Upper Bounds Recall that our goal is to compute Equation 2, which is a sequence of maximizations over the action variables followed by a sequence of summations over the chance variables. Note that maximizations commute, and therefore the maximizations over action variables can be performed in any order. Similarly, the summations over chance variables can be performed in any order. These two sequences cannot be swapped or mixed, though, as maximization does not commute with summation.

However, if we disregard this constraint and allow the maximizations and summations to be mixed in any order, it is not difficult to see that we will get a result that *cannot be lower* than the correct value. The motivation for lifting the variable ordering constraint is that we can now take advantage of the bounded treewidth of these planning problems by performing these maximizations and summations on the compact d-DNNF compilation, and using the results as upper bounds to prune a search.

Specifically, Algorithm 2 computes an upper bound on the success probability of the best completions of a partial plan π, by a single bottom-up traversal of the d-DNNF graph for $\exists S.\Delta$ (note that we are using the same name for the value function as in Algorithm 1, which can now be regarded as a special case of Algorithm 2 where π is a complete plan).

Algorithm 2 Upper Bound

$$val(N,\pi) = \begin{cases} 1, & \text{if } N \text{ is a leaf node that mentions an action variable and is consistent with } \pi; \\ 0, & \text{if } N \text{ is a leaf node that mentions an action variable and is inconsistent with } \pi; \\ Pr(p), & \text{if } N \text{ is a leaf node } p \text{ where } p \text{ is a chance variable;} \\ 1 - Pr(p), & \text{if } N \text{ is a leaf node } \neg p \text{ where } p \text{ is a chance variable;} \\ \prod_i val(N_i,\pi), & \text{if } N = \bigwedge_i N_i; \\ \max_i val(N_i,\pi), & \text{if } N = \bigvee_i N_i \text{ and } N \text{ is a decision node over an action variable not set by } \pi; \\ \sum_i val(N_i,\pi), & \text{if } N = \bigvee_i N_i \text{ and } N \text{ is not of the above type.} \end{cases}$$

ComPlan keeps the d-DNNF graph G on the side during search. For an n-step planning problem, the maximum depth of the search will be n. At each node of the search tree, an unused time step k, $0 \le k < n$, is chosen (this need not be in chronological order), and branches are created corresponding to the different actions that can be taken at step k. The path from the root to the current node is hence a partial plan π, and can be pruned if $val(G,\pi)$ computed by Algorithm 2 is less than or equal to the *lower bound* on the value of an optimal plan. This lower bound is initialized to 0 and updated whenever a leaf is reached and the corresponding complete plan has a greater value. When search terminates, the best plan found together with its value is returned.

Variable Ordering Let $a_k^1, a_k^2, \ldots, a_k^{|\mathbf{A}|}$ be the propositional action variables for step k, where $\mathbf{A}$ is the set of actions. At each node of the search tree, let lb be the current lower bound on the success probability of an optimal plan, let π be the partial plan committed to so far, and let k be some time step that has not been used in π. We are to select a k and branch on the possible actions to be taken at step k. We consider the following:

$$hb_k = \max\{b_i : b_i = val(G, \langle \pi, a_k^i \rangle), b_i > lb\}, \qquad (4)$$

where $\langle \pi, a_k^i \rangle$ denotes the partial plan π extended with one more action a_k^i (and $\overline{a_k^j}$ for all $j \neq i$, implicitly). This quantity hb_k gives the highest value among the upper bounds for the prospective branches that will not be pruned, and we propose to select a k such that hb_k is *minimized*.

The intuition is that the minimization of hb_k gives the *tightest* upper bound on the value of the partial plan π, and by selecting step k as the next branching point, upper bounds subsequently computed are likely to improve as well.

Value Ordering After a time step k is selected for branching, we will explore the unpruned branches a_k^i in *decreasing* order of their upper bounds. The intuition here is that a branch with a higher upper bound is more likely to contain an optimal solution. Discovery of an optimal solution immediately gives the tightest lower bound possible for subsequent pruning, and hence its early occurrence is desirable.

Value Elimination In the process of computing Equation 4 to select k, some branches a_k^i may have been found to be prunable. Although only one k is ultimately selected, all such branches can be pruned as they are discovered. This can be done by asserting $\overline{a_k^i}$ (and adding it to π implicitly) in the d-DNNF graph G for all k and i such that $val(G, \langle \pi, a_k^i \rangle) \le lb$. Upper bounds computed after these assertions will generally improve, because there is now a smaller chance for the first case of Algorithm 2 to execute.

Acknowledgments

National ICT Australia is funded by the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

References

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17:229–264.

Darwiche, A. 2004. New advances in compiling CNF into decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, 328–332.

Darwiche, A. 2005. The C2D compiler user manual. Technical Report D-147, Computer Science Department, UCLA. http://reasoning.cs.ucla.edu/c2d/.

Huang, J. 2006. Combining knowledge compilation and search for conformant probabilistic planning. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS)*.

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 *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI)*, 748–754.

cf2sat and cf2cs+cf2sat: Two Conformant Planners

Héctor Palacios
Departamento de Tecnología
Universitat Pompeu Fabra
Pg Circunvalación, 8
08003 Barcelona, SPAIN
hector.palacios@upf.edu

Abstract

Even under polynomial restrictions on plan length, conformant planning remains a very hard computational problem as plan verification itself can take exponential time. We present two planners for the IPC-5. The first is an optimal complete conformant planner called cf2sat, which transform the PDDL into a propositional theory, that is later compiled into normal form called d–DNNF to obtain a new formula that encodes all the possible plans. This planner gives good results on *pure* conformant problems as emptyroom and sorting-nets, but fails to scale on problems more close to classical planning as bomb-in-the-toilet. Although the heavy price of conformant planning cannot be avoided in general, in many cases conformant plans are verifiable efficiently by means of simple forms of disjunctive inference. We present a second planner cf2cs+cf2sat which is a suboptimal conformant planner that try first to solve a problem by translating it (cf2cs) into an equivalent classical problem, that is then solved by an off-the-shelf classical planner. This translation leads to an efficient but incomplete planner capable of solving non-trivial problems quickly. The translation accommodates simple 'reasoning by cases' by means of an 'split-and-merge' strategy. If cf2cs is not able to solve the problem, cf2cs+cf2sat switch to cf2sat to ensure completeness. Even thought cf2cs is incomplete, it deals successfully with simple problems as bomb-in-the-toilet, and other non-trivial problems as emptyroom.

Introduction

Conformant planning is a form of planning where a goal is to be achieved when the initial situation is not fully known and actions may have non-deterministic effects (Goldman & Boddy 1996)[1]. Conformant planning is computationally harder than classical planning, as even under polynomial restrictions on plan length, plan verification remains hard (Turner 2002).

We present two conformant planners based on two strategies combined properly. The first, cf2sat, is an optimal complete conformant planner that translates the problem into a logic theory, as in SATPLAN (Kautz & Selman 1996). A SAT solver call over this theory would result in one of the possible executions (actions and fluents), which assume a particular initial state, but we want to obtain a plan conformant to all the initial states. From this theory we generate a new one encoding all the possible conformant plans, and call a SAT solver once to obtain a plan. We obtained good results running cf2sat on some very complex domains but failed to scale in more simple problems (Palacios & Geffner 2005).

For this reason we have proposed an incomplete method for mapping from conformant planning to classical planning, cf2cs (Palacios & Geffner 2006). This works by doing limited disjunctive reasoning and allow to solve popular benchmarks like bomb-in-the-toilet, trying to fill the gap left by cf2sat between pure conformant planning and classical planning. The second planner, cf2cs+cf2sat, is a suboptimal complete conformant planning that starts by trying to solve the problem using cf2cs. If it is not possible, the algorithm switch to cf2sat, which is complete.

In the rest of the report we present with more detail the conformant2sat planner and the conformant2classical transformation. Later, we comment about their performance and integration to obtain the presented planners.

Mapping Conformant Planning to SAT

For a conformant planning problem, if the number m of possible initial states $s_0 \in Init$ is bounded (e.g., bounded number of disjunctions of bounded size in the initial situation) and actions are deterministic, the conformant planning problem P with a fixed horizon n can be mapped in the SAT problem over the formula (Palacios & Geffner 2005)

$$\bigwedge_{s_0 \in Init} T^{s0}(P, n) \qquad (1)$$

where $T(P, n)$ is the propositional theory that encodes the problem P with horizon n. $T^{s0}(P, n)$ is $T(P, n)$ with two modifications: first, fluent literals L_0 (L at time 0) are replaced by true/false iff L is true/false in the (complete) state s_0, and second, fluent literals L_i, $i > 0$, are replaced by 'fresh' literals L_i^{s0}, one for each $s_0 \in Init$.

Eq. 1 can be thought as expressing m 'classical planning problems', one for each possible initial state $s_0 \in Init$, that are *coupled* in the sense that they all share the same set of actions; namely, the action variables are the only variables shared across the different subtheories $T^{s0}(P, n)$ for $s_0 \in Init$.

[1]We assume that actions are deterministic and all the uncertainty is on the initial state. This assumption does not lead to loss of expressivity.

However, a planner using Eq. 1 naively will not scale. We have already proposed two approaches to optimal classical conformant planning based on logical formulations (Palacios *et al.* 2005; Palacios & Geffner 2005). Both of them translate the problem into CNF, and obtain a plan by doing logical operations and search. In cf2sat (Palacios & Geffner 2005) (for *conformant2sat*) we construct a new propositional formula:

$$T_{\text{cf}}(P) \quad = \quad \bigwedge_{s_0 \in \mathit{Init}} \textsf{project}[\, T(P) \mid s_0 \;;\; \mathit{Actions}\,] \qquad (2)$$

by doing logical operations as projection (dual of forgetting) and conditioning. The project operation allows to safely *And* over each theory depending on each initial state. The models of the formula $\textsf{project}[\, T(P) \mid s_0 \;;\; \mathit{Actions}\,]$ are the models of $T(P) \mid s_0$ but looking only at the action variables.

Theorem 1 (Palacios & Geffner, 2005) *The models of $T_{cf}(P)$ in (2) are one-to-one correspondence with the conformant plans for the problem P.*

We feed $T_{\text{cf}}(P)$ into a SAT solver to obtain a plan. Logical operations became feasible by compiling the propositional theory into d–DNNF (Darwiche 2002), a formal norm akin to OBDD. The result of compiling a propositional theory ϕ to d–DNNF is a logical circuit that encodes all the possible models of ϕ. Summarizing, the cf2sat algorithm is:

- The following operations are repeated starting from a planning horizon $N = 0$ which is increased by 1 until a solution is found[2].
  1. The theory $T(P)$ is **compiled** into the d–DNNF theory $T_{\text{c}}(P)$
  2. From $T_{\text{c}}(P)$, the transformed theory
     $$T_{\text{cf}}(P) \quad = \quad \bigwedge_{s_0 \in \mathit{Init}} \textsf{project}[\, T_{\text{c}}(P) \mid s_0 \;;\; \mathit{Actions}\,]$$
     is obtained by operations that are linear in time and space in the size of the DAG representing $T_{\text{c}}(P)$.
  3. The theory $T_{\text{cf}}(P)$ is converted into CNF and the **SAT solver** is called upon it.

The plan obtained can be optimal in terms of the number of actions if we forbidden the concurrent execution of every pair of actions. This is known as the *sequential* setting. If we allow non-interfering actions to be executed simultaneously, *parallel* setting, the total executing time or makespan will be minimized.

Actually, it is not necessary to do projection and conditioning for every initial state. By compiling the theory $T(P)$ doing case analysis first on the variables of initial state, we can obtain each $\textsf{project}[\, T(P) \mid s_0 \;;\; \mathit{Actions}\,]$ as a sub-circuit of $T_{\text{cf}}(P)$. Therefore, Eq. 2 can be obtained in linear time over the compiled theory $T_{\text{cf}}(P)$. Translation from this new circuit into CNF is done by introducing propositional variables for each gate and adding clauses to encode the relation between them.

[2]A better lower bound can be the length of an optimal classical plan for one initial state

Compiling uncertainty away: Conformant to Classical Planning (sometimes)

The main motivation of cf2cs is to narrow the gap between conformant planning and classical planning by developing an approach that targets 'simple' conformant problems effectively. The approach is not complete but it provides solutions to non-trivial problems. For instance, simple rules suffice to show that a robot that moves n times to the right in an empty grid of size n, will necessarily end up in the rightmost column.

We have proposed to solve *some non-trivial conformant planning problems* by translating them intro *classical planning problems* (Palacios & Geffner 2006). New problems are fed into a off-the-shelf classical planner. The translation is sound as the classical plans are all conformant, but it is incomplete as the converse does not always hold. The translation scheme accommodates 'reasoning by cases' by means of an 'split-and-merge' strategy; namely, atoms L/X_i that represent conditional beliefs 'if X_i then L' are introduced in the classical encoding that are then combined by suitable actions when certain invariants in the plan are verified.

The translation scheme maps a conformant planning problem P into a classical planning problem $K(P)$. For each atom a in P we add to $K(P)$ new atoms Ka and $K\neg a$. At time t if $Ka \wedge \neg K\neg a$ (resp. $\neg Ka \wedge K\neg a$) holds, then a is true (resp. false) in all the states of the belief state. The initial state of $K(P)$ indicates the atoms that are known to be true or false in the initial belief state of P. Otherwise it states that the value of those atoms is unknown: $\neg Ka \wedge \neg K\neg a$. The goal of P is assumed to be a list of atoms $\{g_1, \ldots, g_n\}$. Therefore, the goal of $K(P)$ requires all those atoms to be known: $\{Kg_1, \ldots, Kg_n\}$. This encoding is related to 0-approximation (Baral & Son 1997). In general, it allow to capture that after doing some actions, the effect can be unsure if the real value of the conditions is not known.

This encoding, so far, does not allow any kind of disjunctive reasoning. We extend the translation further so that the disjunctions in P are taken into account in a form that is similar to the Disjunction Elimination inference rule used in Logic

$$\text{If } X_1 \vee \cdots \vee X_n,\ X_1 \supset L, \ldots, X_n \supset L \text{ then } L \qquad (3)$$

For doing this, we add to $K(P)$ atoms L/X_i to encode that $L \supset X_i$ holds. For example, if for problem P we have the disjunction $X_1 \vee \cdots \vee X_n$ in the initial state, and actions $a_1, \ldots, a_n$ with conditional effect $A \wedge X_i \rightarrow L$; In $K(P)$ those actions will have also conditional effect $A \rightarrow L/X_i$. Informally, $A \rightarrow L/X_i$ can be read as: "If we apply a_i when A is true, we conclude that L is true if X_i is true"[3]. After applying every action a_i, if some invariants were preserved, we can conclude L because $L/X_1 \wedge \cdots \wedge L/X_n$ holds. To allow this conclusion, we add to $K(P)$ a new action $merge_{X,L}$ with conditional effect $L/X_1 \wedge \cdots \wedge L/X_n \rightarrow KL$.

These rules more detailed and other rules can be read in (Palacios & Geffner 2006). They yield expressivity without sacrificing efficiency, as they manage to *accommodate*

[3]It is true if a_i does not modify X_i. In general it is more subtle. More details on (Palacios & Geffner 2006)

non-trivial forms of disjunctive inference in a classical theory without having to carry disjunctions explicitly in the belief state: some disjunctions are represented by atoms like L/X_i, and others are maintained as *invariants* enforced by the resulting encoding.

Theorem 2 (Soundness $K(P)$**)** *(Palacios & Geffner, 2006) Any plan that achieves the literal* KL *in* $K(P)$ *is a plan that achieves* L *in the conformant problem* P.

Results

We ran the optimal planner cf2sat with the Darwiche's d–DNNF compiler c2d v2.18 (Darwiche 2004) and the SAT solver `siege_v4`, obtaining very good results on problems as Emptyroom, Cube-Center, Ring And Sorting-Nets. In general, the compiling step was not the bottleneck. It was not the case in domains like Bomb-in-the-Toilet, where the big number of objects lead to huge theories impossible to be compiled. A middle case was the Ring domain, which lead to big d–DNNFs but later they were very easy for the SAT solver.

We also ran the translator cf2cs from conformant to classical planning on domains where it was able to work, as Emptyroom, Cube-Center, Bomb-in-the-Toilet, Safe, Grid, Logistics. Then we solve those new classical instances by calling the FF (Hoffmann & Nebel 2001) classical planner. Among the popular benchmarks, there are three domains, Sorting-Nets, (Incomplete) Blocks, and Ring, which cannot be handled by this translation scheme. The results were excellent. We were surprised to obtain in general optimal plans even though FF is a suboptimal planner. An interesting point is that the instances resulting of cf2cs have actions with many conditional effects, and many planners available were not able to deal with these instances.

All the relevant programs were written in C++:

- For cf2sat
  - Translator from PDDL to CNF, cconf. It was written by Blai Bonet in joint work (Palacios *et al.* 2005).
  - Translator from $T_{\text{cf}}(P)$, in d–DNNF, to CNF.
- For cf2cs
  - Translator from a PDDL of conformant problem to a PDDL of the equivalent classical problem. The parser was taken from cconf.

Planners for the IPC-5

For the IPC-5, we present two complete planners.

- cf2sat: An optimal parallel conformant planning, using the d–DNNF compiler c2d v2.20 (Darwiche 2004) and the SAT solver `siege_v4`, by Lawrence Ryan, or *zChaff* (Moskewicz *et al.* 2001)[4].
- cf2cs(FF)+cf2sat: A suboptimal conformant planning. It starts trying to solve the problem with cf2cs(FF). If not possible to try with cf2cs(FF) or not solution is found, cf2cs(FF)+cf2sat switch to cf2sat.

[4]siege_v4 was reported to be fast on planning theories (Kautz 2004). Our experiments confirmed that affirmation. Sometimes the CNFs are too big for siege_v4. On these case we try with solve the instances with *zChaff* which is slower in general for our theories.

References

Baral, C., and Son, T. C. 1997. Approximate reasoning about actions in presence of sensing and incomplete information. In *Proc. ILPS 1997*, 387–401.

Darwiche, A. 2002. On the tractable counting of theory models and its applications to belief revision and truth maintenance. *J. of Applied Non-Classical Logics*.

Darwiche, A. 2004. New advances in compiling cnf into decomposable negation normal form. In *Proc. ECAI 2004*, 328–332.

Goldman, R. P., and Boddy, M. S. 1996. Expressive planning and explicit knowledge. In *Proc. AIPS-1996*.

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

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of AAAI-96*, 1194–1201. AAAI Press / MIT Press.

Kautz, H. 2004. Satplan04: Planning and satisfiability. In *Proceedings of IPC-4*.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the 38th Design Automation Conference (DAC'01)*.

Palacios, H., and Geffner, H. 2005. Mapping conformant planning to sat through compilation and projection. In *Spanish Conference on Artificial Inteligence (CAEPIA)*.

Palacios, H., and Geffner, H. 2006. Compiling uncertainty away: Solving conformant planning problems using a classical planner (sometimes). Technical report. Submitted to AAAI-06.

Palacios, H.; Bonet, B.; Darwiche, A.; and Geffner, H. 2005. Pruning conformant plans by counting models on compiled d-DNNF representations. In *Proc. of the 15th Int. Conf. on Planning and Scheduling (ICAPS-05)*. AAAI Press.

Turner, H. 2002. Polynomial-length planning spans the polynomial hierarchy. In *JELIA '02: Proc. of the European Conference on Logics in AI*, 111–124. Springer-Verlag.

The Factored Policy Gradient planner (IPC-06 Version)

Olivier Buffet and **Douglas Aberdeen**
National ICT Australia & The Australian National University
Canberra, Australia
firstname.lastname@nicta.com.au

Abstract

We present the Factored Policy Gradient (FPG) planner: a probabilistic temporal planner designed to scale to large planning domains by applying two significant approximations. Firstly, we use a "direct" policy search in the sense that we attempt to directly optimise a parameterised plan using gradient ascent. Secondly, the policy is factored into a per action mapping from a partial observation to the probabilility of executing, reflecting how desirable each action is. These two approximations — plus memory use that is independent of the number of states — allow us to scale to significantly larger planning domains than were previously feasible. Unlike other probabilistic temporal planners, FPG can attempt to optimise *both* makespan and the probability of reaching the goal. The version of FPG used in the IPC-06 competition optimises the makespan only, and turns off concurrent planning.

Introduction

To date, only a few planning tools have attempted to handle general probabilistic temporal planning domains. These tools have only been able to produce good or optimal policies for relatively small or easy problems. We designed the Factored Policy Gradient (FPG) planner with the goal of creating tools that produce good policies in real-world domains rather than perfect policies in toy domains. We achieve this by: 1) using gradient ascent for direct policy search; 2) factoring the policy into simple approximate policies for starting each action; 3) presenting each policy with critical observations instead of the entire state (implicitly aggregating similar states); and 4) using Monte-Carlo style algorithms with memory requirements that are independent of the state space size.

The AI planning community is familiar with the value-estimation class of reinforcement learning (RL) algorithms, such as RTDP (Barto, Bradtke, & Singh 1995), and arguably AO*. These algorithms represent probabilistic planning problems as a state space and estimate the long-term value, utility, or cost of choosing each action from each state (Mausam & Weld 2005; Little, Aberdeen, & Thiébaux 2005). The fundamental disadvantage of such algorithms is the need to estimate the values of a huge number of state-action pairs. Even algorithms that prune most states still fail to scale due to the exponential increase of relevant states as the domains grow.

On the other hand, the FPG planner borrows from Policy-Gradient reinforcement learning. This class of algorithms does not estimate planning state-action values. Instead, policy-gradient RL algorithms estimate the gradient of the unique long-term average reward of the process. In the context of stochastic shortest path problems, such as the IPC-06 domains, we can view this as estimating the gradient of long-term value of only the initial state. Gradients are computed with respect to the parameters governing the choice of actions at each decision point. These parameters summarise the policy, or plan, of the system. Stepping the parameters in the direction given by the gradient increases the expected return, or value from the initial state. Specifically, we will use the OLPOMDP policy-gradient RL algorithm described by Baxter, Bartlett, & Weaver (2001).

The policy takes the form of a parameterised function that accepts a description of the planning state as input, and returns a probability distribution over legal actions. In our temporal planning setting, an *action* is defined as a single *grounded* durative-action (in the PDDL 2.1 sense).

We factor the parameterised policy by using a function approximator for each action. Only when an action's preconditions are satisfied do we evaluate the desirability (as a probability of executing at this decision point) of the action. By doing this, the number of policy parameters grows linearly with the number of actions and predicates. Our parameterised action policy is a simple multi-layer perceptron that takes the truth value of the predicates at the current planning state, and outputs a probability distribution over whether to start the action. We require that the truth value of the predicates be a good (but not necessarily complete) indicator of the total state of the plan so far.

Background

Input Language

FPG's input language is the complete language handled by `mdpsim`, i.e., PDDL with some minor extensions (Younes & Littman 2004; Younes *et al.* 2005). Indeed, FPG is using mdpsim's data structures and functions to implement the planning domain simulator.

POMDP Formulation of Planning

We deliberately use factored policies that only consider partial state information. Policy gradient methods still converge under partial observability, but their value-based counterparts may not (Singh, Jaakkola, & Jordan 1994).

A finite partially observable Markov decision process consists of: a finite set of states $s \in \mathcal{S}$; a finite set of actions $a \in \mathcal{A}$; probabilities $\mathbb{P}[s'|s,a]$ of making state transition $s \to s'$ under action a; a reward for each state $r(s) : \mathcal{S} \to \mathbb{R}$; and a finite set of observation vectors $\mathbf{o} \in \mathcal{O}$ seen by action policies in lieu of complete state descriptions. FPG draws observations deterministically given the state, but more generally observations may be stochastic. *Goal states* occur when the predicates match a goal state specification. From *failure states* it is impossible to reach a goal state, usually because time or resources have run out. These two classes of state combine to form the set of *terminal* states that produce an immediate reset to the initial state s_0. A single trajectory through the state space, used to estimate gradients, consists of concatenated simulated plan executions that reset to s_0 when a terminal state is reached.[1]

Policies are stochastic, mapping observation vectors $\mathbf{o}$ to a probability over actions. For FPG, an action a is an integer in $[1, N]$, where N is the number of available grounded actions. The probability of action a is $\mathbb{P}[a|\mathbf{o}, \boldsymbol{\theta}]$, where conditioning on $\boldsymbol{\theta}$ reflects the fact that the policy is controlled by a set of real valued parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. The maximum makespan of a plan is limited, ensuring that all stochastic policies reach reset states in finite time when executed from s_0.

The GPOMDP algorithm maximises the long-term average reward

$$R(\boldsymbol{\theta}) := \lim_{T\to\infty} \frac{1}{T} \mathbb{E}_{\boldsymbol{\theta}} \left[\sum_{t=1}^{T} r(s_t) \right], \tag{1}$$

where the expectation $\mathbb{E}_{\boldsymbol{\theta}}$ is over the distribution of state trajectories $\{s_0, s_1, \dots\}$ induced by $\boldsymbol{P}(\boldsymbol{\theta})$. In the context of planning, the instantaneous reward provides the action policies with a measure of progress toward the goal. Our simple reward scheme is to set $r(s) = 1000$ for all states s that represent the goal state, and 0 for all other states. To maximise $R(\boldsymbol{\theta})$, *goal* states must be reached as frequently as possible. This has the desired property of simultaneously minimising plan duration and maximising the probability of reaching the goal (failure states achieve no reward).

Planning State Space

As already mentionned, this implementation of FPG is using `mdpsim`'s data structures and functions (Younes *et al.* 2005). Thus, a state includes all true predicates as well as the value of each function. The observation vector used by FPG needs to have one entry for each predicate that could be true at some point. Thus, a first step (once the problem has been loaded) is to generate these predicates, which is done simultaneously when `mdpsim` grounds actions.

To estimate gradients we need a plan execution simulator to generate a trajectory through the planning state space. Here again, FPG's simple solution is to use `mdpsim`'s next() function, which samples a next state s' given current state s and chosen action a.

[1]This concatenation trick is simply used to formulate the SSP planning in the framework used in Baxter, Bartlett, & Weaver (2001). In practice we can take advantage of the episodic nature of the problem.

Policy-Gradient Reinforcement Learning

Each action a determines a stochastic matrix $\boldsymbol{P}(a) = [\mathbb{P}[s'|s,a]]$ of transition probabilities from state s to state s' given action a. The gradient estimator discussed in this paper does not assume explicit knowledge of $\boldsymbol{P}(a)$ or of the observation process.

All policies are stochastic, with a probability of choosing action a given state s, and parameters $\boldsymbol{\theta} \in \mathbb{R}^n$ of $\mathbb{P}[a|\mathbf{o}, \boldsymbol{\theta}]$. During the course of optimisation the policy becomes more deterministic. The evolution of the state s is Markovian, governed by an $|\mathcal{S}| \times |\mathcal{S}|$ transition probability matrix $\boldsymbol{P}(\boldsymbol{\theta}) = [\mathbb{P}[s'|\boldsymbol{\theta}, s]]$ with entries given by

$$\mathbb{P}[s'|\boldsymbol{\theta}, s] = \sum_{a\in\mathcal{A}} \mathbb{P}[a|\mathbf{o}, \boldsymbol{\theta}]\, \mathbb{P}[s'|s,a]\,. \tag{2}$$

GPOMDP is an infinite-horizon policy-gradient method to compute the gradient of the *long-term average reward* (1) with respect the policy parameters $\boldsymbol{\theta}$. In this abstract we give only the gradient estimator customised for planning. For the derivation of the gradient estimator, and proofs of convergence, please refer to Baxter, Bartlett, & Weaver (2001).

Policy-Gradient for Planning

The desirability of some eligible action i in the set of actions with satisfied preconditions E_t is a real value $d(i)$ computed by a multi-layer perceptron. This preceptron usually has at most one hidden layer, and its weight vector $\boldsymbol{\theta}_i$ is part of the complete vector of parameters $\boldsymbol{\theta}$ learned by reinforcement. With input vector $\mathbf{o}$, the perceptron computes $d(i) = f_i(\mathbf{o}_t; \boldsymbol{\theta}_i)$.

Action a_t is sampled from a probability distribution obtained by computing a Gibbs[2] distribution over $d(i)$'s of eligible actions as follows:

$$\mathbb{P}[a_t = i|\mathbf{o}_t, \boldsymbol{\theta}] = \frac{exp(f_i(\mathbf{o}_t; \boldsymbol{\theta}_i))}{\sum_{j\in E_t} exp(f_j(\mathbf{o}_t; \boldsymbol{\theta}_j))}. \tag{3}$$

Initially, the parameters are set to small random values giving a near uniform random policy; encouraging exploration of the action space. Each gradient step typically moves the parameters closer to a deterministic policy. After some experimentation we chose an observation vector that is a binary description of predicates current truth values plus a constant 1 bit to provide bias to the preceptron.

The observations are simply the truth value of the predicates for the current state.

[2]Essentially the same as a Boltzmann or soft-max distribution.

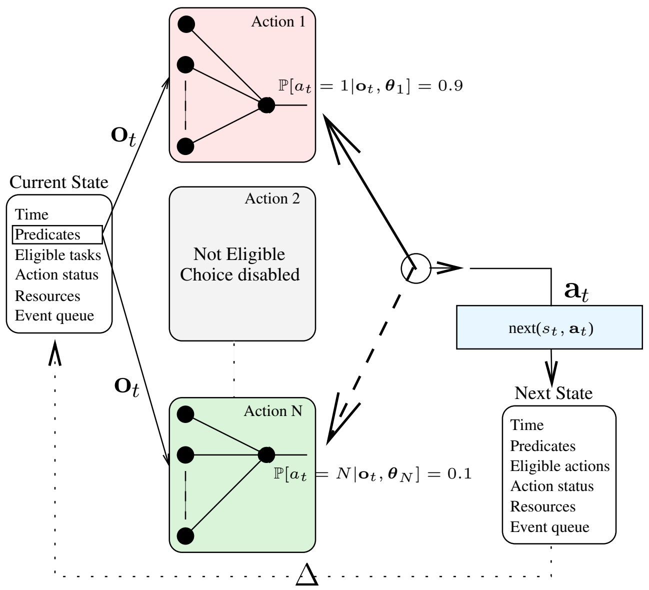

Fig. 1: Individual action-policies make independent decisions.

The FPG Gradient Estimator

Alg. 1 completes our description of FPG by showing how to implement OLPOMDP for planning with factored action policies. The algorithm works by sampling a single long trajectory through the state space: 1) the first state represents time 0 in the plan; 2) the perceptrons attached to eligible actions all receive the vector observation $\mathbf{o}_t$ of the current state s_t; 3) each network computes the desirability of its action; 4) a planning action is sampled; 5) the state transition is sampled; 6) the planner receives the global reward for the new state action and produces an instantaneous gradient $\mathbf{g}_t = r_t\mathbf{e}_t$; 7) all parameters are immediately updated in the direction of $\mathbf{g}_t$.

Algorithm 1 OLPOMDP FPG Gradient Estimator

```
 1: Set $s_0$ to initial state, $t = 0, \mathbf{e}_t = [0]$, init $\boldsymbol{\theta}_0$ randomly
 2: while $R$ not converged do
 3:     $\mathbf{e}_t = \beta\mathbf{e}_{t-1}$
 4:     Extract predicate values as observation $\mathbf{o}_t$ of $s_t$
 5:     for Each eligible action $i$ do
 6:         Evaluate desirability $d(i) = f_i(\mathbf{o}_t; \boldsymbol{\theta}_{ti})$
 7:     Sample action $i$ with probability $\mathbb{P}[a_t = i|\mathbf{o}_t; \boldsymbol{\theta}_t]$
 8:     $\mathbf{e}_t = \mathbf{e}_{t-1} + \nabla \log \mathbb{P}[a_t|\mathbf{o}_t; \boldsymbol{\theta}_t]$
 9:     $s_{t+1}$ = next($s_t, \mathbf{a}_t$)
10:     $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha r_t \mathbf{e}_{t+1}$
11:     if $s_{t+1}$.isTerminalState then $s_{t+1} = s_0$
12:     $t \leftarrow t + 1$
```

Because planning is inherently episodic we could alternatively set $\beta = 1$ and reset $\mathbf{e}_t$ every time a terminal state is encountered. However, empirically, setting $\beta = 0.9$ performed better than resetting $\mathbf{e}_t$.[3] The gradient for parameters not relating to eligible actions is 0. We do not compute $f_i(\mathbf{o}_t; \boldsymbol{\theta}_i)$ or gradients for actions with unsatisfied preconditions. Line 11 resets to the initial planning state upon encountering a terminal state.

[3]Perhaps because $\beta < 1$ can reduce the variance of gradient estimates, even in the episodic case.

Conclusion

FPG diverges from traditional planning approaches in two key ways: we search for plans directly, using a local optimisation procedure (an on-line gradient ascent); and we simplify the plan representation by factoring the plan into a function approximator for each action, each of which observes only a stripped down version of the current state.

The drawback of our approach is that local optimisation, simplified parameterisations, and reduced observability can all lead to sub-optimal plans; but this sacrifice is deliberate in order to achieve scalability through memory use and *per step* computation times that grow linearly with the domain. However, the total number of gradient steps is a function of the *mixing time* of the underlying POMDP, which can grow exponentially with the state variables. How to compute the mixing time of an arbitrary MDP is an open problem. This hints at the hardness of assessing in advance the difficulty of general planning problems.

FPG is a planner with great potential to produce *good* policies in large domains, especially considering the version handling concurrency. Further work will refine our parameterised action policies, apply more sophisticated stochastic gradient ascent methods, and attempt to characterise possible local minima.

Acknowledgments

National ICT Australia is funded by the Australian Government's Backing Australia's Ability program and the Centre of Excellence program. This project was also funded by the Australian Defence Science and Technology Organisation.

References

Barto, A.; Bradtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72.

Baxter, J.; Bartlett, P.; and Weaver, L. 2001. Experiments with infinite-horizon, policy-gradient estimation. *JAIR* 15:351–381.

Little, I.; Aberdeen, D.; and Thiébaux, S. 2005. Prottle: A probabilistic temporal planner. In *Proc. AAAI'05*.

Mausam, and Weld, D. S. 2005. Concurrent probabilistic temporal planning. In *Proc. International Conference on Automated Planning and Scheduling*. Moneteray, CA: AAAI.

Singh, S.; Jaakkola, T.; and Jordan, M. 1994. Learning without state-estimation in partially observable Markovian decision processes. In *Proceedings of ICML 1994*, number 11.

Younes, H. L. S., and Littman, M. L. 2004. PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. Technical Report CMU-CS-04-167, Carnegie Mellon University.

Younes, H. L. S.; Littman, M. L.; Weissman, D.; and Asmuth, J. 2005. The first probabilistic track of the international planning competition. *Journal of Artificial Intelligence Research* 24:851–887.

Paragraph: A Graphplan-based Probabilistic Planner

Iain little
National ICT Australia & Computer Sciences Laboratory
The Australian National University
Canberra, ACT 0200, Australia

Introduction

`Paragraph` is a probabilistic planner that finds contingency plans that maximise the probability of reaching the goal within a given time horizon. It is capable of finding either a cyclic or acyclic solution to a given problem, depending on how it is configured. These solutions are optimal in the non-concurrent case, and optimal for a restricted model of concurrency. The concurrent case is not relevant to this discussion, and is not further discussed. A detailed description of `Paragraph` is given in (Little & Thiébaux 2006).

The `Graphplan` framework (Blum & Furst 1997) is an approach that has proven to be highly successful for solving classical planning problems. Extensions of this framework for probabilistic planning have been developed (Blum & Langford 1999), but either dispense with the techniques that enable concurrency to be efficiently managed, or are unable to produce optimal contingency plans. Specifically, `PGraphplan` finds optimal (non-concurrent) contingency plans via dynamic programming, using information propagated backwards through the planning graph to identify states from which the goal is provably unreachable. This approach takes advantage of neither the state space compression inherent in `Graphplan`'s goal regression search, nor the pruning power of `Graphplan`'s mutex reasoning and nogood learning. `TGraphplan` is a minor extension of the original `Graphplan` algorithm that computes a single path to the goal with a maximal probability of success; replanning could be applied when a plan's execution deviates from this path, but this strategy is not optimal.

`Paragraph` is an extension of the `Graphplan` algorithm to probabilistic planning. It enables much of the existing research into the `Graphplan` framework to be transfered to the probabilistic setting. `Paragraph` is a planner that implements some of this potential, including: a probabilistic planning graph, powerful mutex reasoning, nogood learning, and a goal regression search. The key to this framework is an efficient method of finding optimal contingency plans using goal regression. This method is fully compatible with the `Graphplan` framework, but is also more generally applicable.

Algorithm

To extend the `Graphplan` framework to the probabilistic setting, it is necessary to extend the planning graph data structure to account for uncertainty. We do this by introducing a node for each of an action's possible outcomes, so that there are three different types of nodes in the graph: proposition, action, and outcome. Action nodes are then linked to their respective outcome nodes, and edges representing effects link outcome nodes to proposition nodes. Each persistence action has a single outcome with a single add effect. We refer to a persistence action's outcome as a *persistence outcome*. This extension is functionally equivalent to that described in (Blum & Langford 1999), except that we also adapt the planning graph's mutex propagation rules from the deterministic setting.

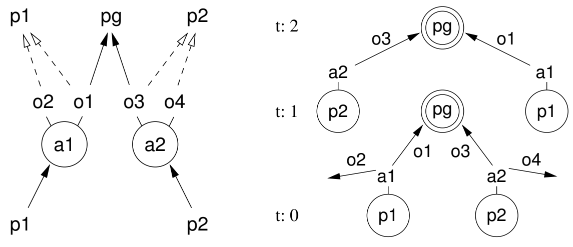

Figure 1: An action-outcome-proposition dependency graph and search space for an example problem.

The solution extraction step of the `Graphplan` algorithm relies on a backward search through the structure of the planning graph. In classical planning, the goal is to find a subset of action nodes for each level such that the respective sequence of action sets constitutes a valid trajectory. The search starts from the final level of the graph, and attempts to extend partial trajectories one level at a time until a solution is found.

`Paragraph` uses this type of goal-regression search with an explicit representation of the expanded search space. This search is applied exhaustively, to find all trajectories that the `Graphplan` algorithm can find. An optimal contingency plan is formed by linking these trajectories together. This requires some additional computation, and involves using forward simulation through the search space to compute the possible world states at reachable search nodes.

As observed by Blum and Langford (1999), the difficulty with combining probabilistic planning with `Graphplan`-style regression is in correctly and efficiently combining the trajectories. Sometimes the trajectories will 'naturally' join together during the regression, which happens when search nodes share a predecessor through different 'joint outcomes' (sets of outcomes) of the same action set.

Unfortunately, the natural joins are not sufficient to find all contingencies. Consider the problem shown in Figure 1, which we define as:[1] the propositions $p1$, $p2$ and pg; $s_0 = \{p1, p2\}$; $G = \{pg\}$; the actions $a1$ and $a2$; and the outcomes $o1$ to $o4$. $a1$ has precondition $p1$ and outcomes $\{o1, o2\}$; $a2$ has precondition $p2$ and outcomes $\{o3, o4\}$. Both actions always delete their precondition; $o1$ and $o3$ both add pg. The optimal plan for this example is to execute one of the actions; if the first action does not achieve the goal, then the other action is executed.

The backward search will correctly recognise that executing $a1$–$o1$ or $a2$–$o3$ will achieve the goal, but it fails to realise that $a1$–$o2, a2$–$o3$ and $a2$–$o4, a1$–$o1$ are also valid trajectories. The longer trajectories are not discovered because they contain a 'redundant' first step; there is no way of relating the effect of $o2$ and the precondition of $a2$, or the effect of $o4$ with the precondition of $a1$. While these undiscovered trajectories are not the most desirable execution sequences, they are necessary for an optimal contingency plan. In classical planning, it is actually a good thing that trajectories with this type of redundancy cannot be discovered, as redundant steps only hinder the search for a single shortest trajectory. Identifying the missing trajectories requires some additional computation beyond the goal regression search. We refer to trajectories that can be found using unadorned goal regression as *natural trajectories*.

The solution we have developed is based on constructing all 'non-redundant' contingency plans by linking together the trajectories that goal regression is able to find. This is sufficient to find an optimal solution, as there always exists at least one non-redundant optimal plan. `Paragraph` combines pairs of trajectories by linking a node in one trajectory to a node in the other. This can be done when a possible world state of the earlier node has a resulting world state that subsumes the goal set of the later node.

A detailed description of `Paragraph`'s acyclic search algorithm follows.The first step is to generate a planning graph from the problem specification. This graph is expanded until all goal propositions are present and not mutex with each other, or until the graph levels off to prove that no solution exists. Assuming the former case, a depth-first goal regression search is performed from a goal node for the graph's final level. This search exhaustively finds all natural trajectories from the initial conditions to the goal. Once this search has completed, the possible world states for each trajectory node are computed by forward-propagation from time 0, and the node/state costs are updated by backward-propagation from the goal node. Potential trajectory joins are detected each time a new node is encountered during the backward search, and each time a new world state is computed during the forward state propagation. Unless a termination condition has been met, the planning graph is then expanded by a single level, and the backward search is performed from a new goal node that is added to the existing search space. This alternation between backward search, state simulation, cost propagation, and graph expansion continues until a termination condition is met. An optimal contingency plan is then extracted from the search space by traversing the space in the forward direction using a greedy selection policy.

[1]This problem was used by Blum and Langford (1999) to illustrate the difficulty of using goal-regression for probabilistic planning, and to explain their preference of a forward search in `PGraphplan`.

Classical planning problems have the property that the shortest solution to a problem will not visit any given world state more than once. This is no longer true for probabilistic planning, as previously visited states can unintentionally be returned to by chance. Because of this, it is common for probabilistic planners to allow for cyclic solutions. An overview of the algorithm for producing such solutions follows. This method departs further from the `Graphplan` algorithm than the acyclic search does: fundamental to the `Graphplan` algorithm is a notion of time, which we dispense with for `Paragraph`'s cyclic search.

The cyclic search does not preserve `Graphplan`'s alternation between graph expansion and backward search: the planning graph is expanded until it levels off, and only then is the backward search performed. As there is no notion of time, the backward search is constrained only by the information represented in the final level of the levelled-off planning graph.

This cyclic search uses either a depth-first or iterative deepening algorithm. In both cases, the search uses the outcomes supporting the planning graph's final level of propositions when determining a search node's predecessors. The same principal is applied to nogood pruning: only the mutexes in the final level of the planning graph—those that are independent of time—can be safely used. An important consequence of only using universally applicable nogoods is that any new nogoods learnt from `failure` nodes are also universal. Neither search strategy is clearly superior. The depth-first search is usually preferable when searching the entire search space, as it is more likely to learn useful nogoods. A consequence of this is that there is no predictable order in which the trajectories are discovered. In contrast, the iterative deepening search will find the shortest trajectories first, which can be advantageous when only a subset of the search space might be explored.

References

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

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

Little, I., and Thiébaux, S. 2006. Concurrent probabilistic planning in the graphplan framework. In *Proc. ICAPS*.

Probabilistic Planning via Linear Value-approximation of First-order MDPs

Scott Sanner
University of Toronto
Department of Computer Science
Toronto, ON, M5S 3H5, CANADA
ssanner@cs.toronto.edu

Craig Boutilier
University of Toronto
Department of Computer Science
Toronto, ON, M5S 3H5, CANADA
cebly@cs.toronto.edu

Abstract

We describe a probabilistic planning approach that translates a PPDDL planning problem description to a first-order MDP (FOMDP) and uses approximate solution techniques for FOMDPs to derive a value function and corresponding policy. Our FOMDP solution techniques represent the value function linearly w.r.t. a set of first-order basis functions and compute suitable weights using *lifted*, first-order extensions of approximate linear programming (FOALP) and approximate policy iteration (FOAPI) for MDPs. We additionally describe techniques for automatic basis function generation and decomposition of universal rewards that are crucial to achieve autonomous and tractable FOMDP solutions for many planning domains.

From PPDDL to First-order MDPs

It is straightforward to translate a PPDDL [12] planning domain into the situation calculus representation used for first-order MDPs (FOMDPs); the primary part of this translation requires the conversion of PPDDL action schemata to *effect axioms* in the situation calculus, which are then compiled into *successor-state axioms* [8] used in the FOMDP description. In the following algorithm description, we will assume that we are given a FOMDP specification and we will describe techniques for approximating its value function linearly w.r.t. a set of first-order basis functions. From this value function it is straightforward to derive a first-order policy representation that can be used for action selection in the original PPDDL planning domain.

Linear Value Approximation for FOMDPs

The following explanation assumes the reader is familiar with the FOMDP formalism and operators used in Boutilier, Reiter and Price [2] and extended by Sanner and Boutilier [9]. In the following text, we will refer to function symbols $A_i(\vec{x})$ that correspond to parameterized actions in the FOMDP; for every action and fluent, we expect that a successor state axiom has been defined. The reader should be familiar with the notation and use of the $rCase$, $vCase$, and $pCase$ *case statements* for representing the respective FOMDP reward, value, and transition functions. The reader should also be familiar with the *case operators* $\oplus$, $\ominus$, $\cup$, and $Regr(\cdot)$ [2] as well as $FODTR(\cdot)$, $B^{A(\vec{x})}(\cdot)$, and $B^{A}(\cdot)$ [9].

Value Function Representation

Following [9], we represent a value function as a weighted sum of k *first-order basis functions* in case statement format, denoted $bCase_j(s)$, each containing a *small* number of formulae that provide a first-order abstraction of state space:

$$vCase(s) = \oplus_{i=1}^{k} \; w_i \cdot bCase_i(s) \qquad (1)$$

Using this format, we can often achieve a reasonable approximation of the exact value function by exploiting the additive structure inherent in many real-world problems (e.g., additive reward functions or problems with independent subgoals). Unlike exact solution methods where value functions can grow exponentially in size during the solution process and must be logically simplified [2], here we maintain the value function in a compact form that requires no simplification, just discovery of good weights.

We can easily apply the FOMDP backup operator $B^{A(\vec{x})}$ [9] to this representation and obtain some simplification as a result of the structure in Eq. 1. Exploiting the properties of the $Regr$ and $\oplus$ operators, we find that the backup $B^{A(\vec{x})}$ of a linear combination of basis functions is simply the linear combination of the first-order decision-theoretic regression (FODTR) of each basis function [9]:

$$B^{A(\vec{x})}(\oplus_i \; w_i bCase_i(s)) = \qquad (2)$$
$$rCase(s,a) \oplus (\oplus_i \; w_i FODTR(bCase_i(s), A(\vec{x})))$$

A corresponding definition of B^A follows directly [9]. It is important to note that during the application of these operators, we never explicitly ground states or actions, in effect achieving *both state and action space abstraction*.

First-order Approximate Linear Programming

First-order approximate linear programming (FOALP) was introduced by Sanner and Boutilier [9]. Here we present a linear program (LP) with first-order constraints that generalizes the solution from MDPs to FOMDPs:

$$\text{Variables: } w_i \; ; \;\; \forall i \leq k$$
$$\text{Minimize: } \sum_{i=1}^{k} w_i \sum_{\langle \phi_j, t_j \rangle \in bCase_i} \frac{t_j}{|bCase_i|}$$
$$\text{Subject to: } 0 \geq B^{A}_{\max}(\oplus_{i=1}^{k} \; w_i \cdot bCase_i(s))$$
$$\ominus (\oplus_{i=1}^{k} \; w_i \cdot bCase_i(s)) \; ; \; \forall \; A, s \qquad (3)$$

The objective of this LP requires some explanation. If we were to directly generalize the objective for MDPs to that of FOMDPs, the objective would be ill-defined (it would sum over infinitely many situations s). To remedy this, we suppose that each basis function partition is chosen because it represents a potentially useful partitioning of state space, and thus sum over each case *partition*.

This LP also contains a first-order specification of constraints, which somewhat complicates the solution. Before tackling this, we introduce a general *first-order LP* format that we can reuse for approximate policy iteration:

$$\begin{aligned} \text{Variables: } & v_1, \ldots, v_k \ ; \\ \text{Minimize: } & f(v_1, \ldots, v_k) \\ \text{Subject to: } & 0 \geq case_{1,1}(s) \oplus \ldots \oplus case_{1,n}(s) \ ; \ \forall\, s \qquad (4) \\ & \vdots \\ & 0 \geq case_{m,1}(s) \oplus \ldots \oplus case_{m,n}(s) \ ; \ \forall\, s \end{aligned}$$

The variables and objective are as defined in a typical LP, the main difference being the form of the constraints. While there are an infinite number of constraints (i.e., one for every situation s), we can work around this since case statements are finite. Since the value t_i for each case partition $\langle \phi_i(s), t_i \rangle$ is piecewise constant over all situations satisfying $\phi_i(s)$, we can explicitly sum over the $case_i(s)$ statements in each constraint to yield a single case statement. For this "flattened" case statement, we can easily verify that the constraint holds in the finite number of piecewise constant partitions of the state space. However, generating the constraints for each "cross-sum" can yield an exponential number of constraints. Fortunately, we can generalize constraint generation techniques [10] to avoid generating all constraints. We refer to [9] for further details. Taken together, these techniques yield a practical FOALP solution to FOMDPs.

First-order Approximate Policy Iteration

We now turn to a first-order generalization of approximate policy iteration (FOAPI). Policy iteration requires that a suitable first-order policy representation be derivable from the value function $vCase(s)$. Assuming we have m parameterized actions $\{A_1(\vec{x}), \ldots, A_m(\vec{x})\}$, we can represent the policy $\pi Case(s)$ as:

$$\pi Case(s) = \max\Big(\bigcup_{i=1\ldots m} B^{A_i}(vCase(s))\Big) \qquad (5)$$

Here, $B^{A_i}(vCase(s))$ represents the values that can be achieved by any instantiation of the action $A_i(\vec{x})$ and the $\max$ case operator ensures that the highest possible value is assigned to every situation s. For bookkeeping purposes, we require that each partition $\langle \phi, t \rangle$ in $B^{A_i}(vCase(s))$ maintain a mapping to the action A_i that generated it, which we denote as $\langle \phi, t \rangle \to A_i$. Then, given a particular world state s at run-time, we can evaluate $\pi Case(s)$ to determine which policy partition $\langle \phi, t \rangle \to A_i$ is satisfied in s and thus, which action A_i should be applied. If we retrieve the bindings of the existentially quantified action variables in ϕ (recall that B^{A_i} existentially quantifies these), we can easily determine the instantiation of action A_i prescribed by the policy.

For our algorithms, it is useful to define a set of case statements for each action A_i that is satisfied only in the world states where A_i should be applied according to $\pi Case(s)$. Consequently, we define an action restricted policy $\pi Case_{A_i}(s)$ as follows:

$$\pi Case_{A_i}(s) = \{\langle \phi, t \rangle | \langle \phi, t \rangle \in \pi Case(s) \text{ and } \langle \phi, t \rangle \to A_i\}$$

Following the approach to approximate policy iteration for factored MDPs provided by Guestrin *et al* [4], we can generalize approximate policy iteration to the first-order case by calculating successive iterations of weights $w_j^{(i)}$ that represent the best approximation of the fixed point value function for policy $\pi Case^{(i)}(s)$ at iteration i. We do this by performing the following two steps at every iteration i: (1) Obtaining the policy $\pi Case(s)$ from the current value function and weights ($\sum_{j=1}^{k} w_j^{(i)} bCase_j(s)$) using Eq. 5, and (2) solving the following LP in the format of Eq. 4 that determines the weights of the Bellman error minimizing approximate value function for policy $\pi Case(s)$:

$$\begin{aligned} \text{Variables: } & w_1^{(i+1)}, \ldots, w_k^{(i+1)} \\ \text{Minimize: } & \phi^{(i+1)} \qquad (6) \\ \text{Subject to: } & \phi^{(i+1)} \geq \Big| \pi Case_A(s) \oplus \oplus_{j=1}^{k} [w_j^{(i+1)} bCase_j(s)] \\ & \qquad \ominus \oplus_{j=1}^{k} w_j^{(i+1)} (B^A_{\max} bCase_j)(s) \Big| ; \ \forall A, s \end{aligned}$$

We've reached convergence *if* $\pi^{(i+1)} = \pi^{(i)}$. If policy iteration converges, the loss bounds from [4] generalize directly to the first-order case.

Greedy Basis Function Generation

The use of linear approximations requires a good set of basis functions that span a space that includes a good approximation to the value function. While some work has addressed the issue of basis function generation [7; 5], none has been applied to RMDPs or FOMDPs. We consider a basis function generation method that draws on the work of Gretton and Thiebaux [3], who use inductive logic programming (ILP) techniques to construct a value function from sampled experience. Specifically, they use regressions of the reward as candidate building blocks for ILP-based construction of the value function. This technique has allowed them to generate fully or k-stage-to-go optimal policies for a range of Blocks World problems.

We leverage a similar approach for generating candidate basis functions for use in the FOALP or FOAPI solution techniques. If some portion of state space ϕ has value $v > \tau$ in an existing approximate value function for some nontrivial threshold τ, then this suggests that states that can reach this region (i.e., found by Regr(ϕ) through some action) should also have reasonable value. However, since we have already assigned value to ϕ, we want the new basis function to focus on the area of state space not covered by ϕ. Consequently, we negate ϕ and conjoin it with $Regr(\phi)$ yielding the new basis function $[\neg\phi \wedge Regr(\phi) : 1; \ \phi \vee \neg Regr(\phi) : 0]$. The "orthogonality" of newly generated basis functions also allows for computational optimizations since many combinations of basis function partitions are mutually exclusive and thus need not be examined.

Handling Universal Rewards

In first-order domains, we are often faced with *universal reward expressions* that assign some positive value to the world states satisfying a formula of the general form $\forall y\ \phi(y, s)$, and 0 otherwise. For instance, in a logistics problem, we can use a predicate $Dst(t, c)$ to indicate that truck t is at city c and a fluent $TAt(t, c, s)$ to indicate that truck t is at city c in situation s. Then a reward may be given for having all trucks at their assigned destination: $\forall t, cDst(t, c) \rightarrow TAt(t, c, s)$. One difficulty with such rewards is that our basis function approach provides a piecewise constant approximation to the value function (i.e., each basis function aggregates state space into regions of equal value, with the linear combination simply providing constant values over somewhat smaller regions). However, the value function for problems with universal rewards typically depends (often in a linear or exponential way) on the *number* of domain objects of interest. For instance, in our example, value at a state depends on the number of trucks not at their proper destination (since that impacts the time it will take to obtain the reward). Unfortunately, *this cannot be represented* concisely using the piecewise constant decomposition offered by first-order basis functions. As noted by Gretton and Thiebaux [3], effectively handling universally quantified rewards is one of the most pressing issues in the practical solution of FOMDPs.

To address this problem we adopt a decompositional approach, motivated in part by techniques for additive rewards in MDPs [1; 11; 6; 7]. Intuitively, given a goal-oriented reward that assigns positive reward if $\forall yG(y, s)$ is satisfied, and zero otherwise, we can decompose it into a set of ground goals $\{G(\vec{y_1}), \ldots, G(\vec{y_n})\}$ for all possible $\vec{y_j}$ in a ground domain of interest. If we reach a world state where all ground goals are true, then we have satisfied $\forall yG(y, s)$.

Of course, our methods solve FOMDPs without knowledge of the specific domain, so the set of ground goals that will be faced at run-time is unknown. So in the offline solution of the MDP we assume a a *generic* ground goal $G(\vec{y}^*)$ for a "generic" object vector $\vec{y}^*$. It is easy to construct an instance of the reward function $rCase(s)$ for this single goal, and solve for this simplified generic goal using FOALP or FOAPI. This produces a value function and policy that assumes that $\vec{y}^*$ is the only object vector of interest (i.e., satisfying relevant type and preconditions) in the domain. From this, we can also derive the optimal Q-function for the simplified "generic" domain (and action template $A_i(\vec{x})$): $Q_{G(\vec{y}^*)}(A_i(\vec{x}), s) = B^{A_i}(vCase(s))$.[1] Intuitively, given a ground state s, the optimal action for this generic goal can be determined by finding the ground $A_i(\vec{x}^*)$ for this s with max Q-value.

With the solution (i.e., optimal Q-function) of a generic goal FOMDP, we now address the online problem of action selection for a *specific* domain instantiation. Assume a set of ground goals $\{G(\vec{y_1}), \ldots, G(\vec{y_n})\}$ corresponding to a specific domain given at run-time. If we assume that (typed) domain objects are treated uniformly in the uninstantiated FOMDP, as is the case in many logistics and planning problems, then we obtain the Q-function for any goal $G(\vec{y_j})$ by replacing all ground terms $\vec{y}^*$ with the respective terms $\vec{y_j}$ in $Q_{G(\vec{y}^*)}(A_i(\vec{x}), s)$ to obtain $Q_{G(\vec{y_j})}(A_i(\vec{x}), s)$.

Action selection requires finding an action that maximizes value w.r.t. the original universal reward. Following [1; 6], we do this by treating the *sum of the Q-values* of any action in the subgoal MDPs as a measure of its Q-value in the joint (original) MDP. Specifically, we assume that each goal contributes uniformly and additively to the reward, so the Q-function for a entire set of ground goals $\{G(\vec{y_1}), \ldots, G(\vec{y_n})\}$ determined by our domain instantiation is just $\sum_{j=1}^{n} \frac{1}{n} Q_{G(\vec{y_j})}(A_i(\vec{x}), s)$. The action selection (at run-time) in any ground state is realized by choosing that action with maximum joint Q-value. Naturally, we do not want to explicitly create the joint Q-function, but an efficient scoring technique that evaluates potentially useful actions by iterating through the individual Q-functions is very straightforward.

While this additive and uniform decomposition may not be appropriate for all domains with goal-oriented universal rewards, we have found it to be highly effective for the *Box-World* logistics domain from the ICAPS 2004 probabilistic planning competition. And while this approach can only currently handle rewards with universal quantifiers, this reflects the form of many practical planning problems.

[1]Since the B^A operator can often retain much of the additive structure in the linear approximation of $vCase(s)$ [9], representation and computation with this Q-function is *very* efficient.

References

[1] C. Boutilier, R. I. Brafman, and C. Geib. Prioritized goal decomposition of Markov decision processes: Toward a synthesis of classical and decision theoretic planning. *IJCAI-97*, pp.1156–1162, Nagoya, 1997.

[2] C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order MDPs. *IJCAI-01*, 2001.

[3] C. Gretton and S. Thiebaux. Exploiting first-order regression in inductive policy selection. *UAI-04*, 2004.

[4] C. Guestrin, D. Koller, R. Parr, and S. Venktaraman. Efficient solution methods for factored MDPs. *JAIR*, 2002.

[5] S. Mahadevan. Samuel meets amarel: Automating value function approximation using global state space analysis. *AAAI-05*, pp.1000–1005, Pittsburgh, 2005.

[6] N. Meuleau, M. Hauskrecht, K. Kim, L. Peshkin, L. P. Kaelbling, T. Dean, and C. Boutilier. Solving very large weakly coupled Markov decision processes. *AAAI-98*, 1998.

[7] P. Poupart, C. Boutilier, R. Patrascu, and D. Schuurmans. Piecewise linear value function approximation for factored MDPs. *AAAI 02*, pp.292–299, Edmonton, 2002.

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

[9] S. Sanner and C. Boutilier. Approximate linear programming for first-order MDPs. *UAI 2005)*, Edinburgh, 2005.

[10] D. Schuurmans and R. Patrascu. Direct value approximation for factored MDPs. *NIPS-2001*, Vancouver, 2001.

[11] S. P. Singh and D. Cohn. How to dynamically merge Markov decision processes. *NIPS-98*, 1998.

[12] H. Younes and M. Littman. PPDDL: The probabilistic planning domain definition language, 2004.

Symbolic Stochastic Focused Dynamic Programming with Decision Diagrams

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 2006 International Planning Competition. The planner transforms the PPDDL problems into factored MDPs that are then solved with a structured modified value iteration algorithm based on the safest stochastic path computation from the initial states to the goal states. First, a state subspace is computed by making all the transitions deterministic. Then, a step of modified value iteration on the current reachable state subspace alternates with a step of reachable state expansion by following the current policy.

Introduction

Co-located with the 16th International Conference on Automated Planning and Scheduling, the probabilistic planning track of the 5th International Planning Competition aims at evaluating and at motivating research in the field of non-determinism in planning (Bonet & Givan 2005). In this article, we present a planner based on Markov Decision Processes (MDPs) (Puterman 1994) that have become a popular stochastic framework for planning under uncertainty: the uncertain effects of actions are modeled in a decision-theoretic framework.

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 $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$ where S is the set of states, A is the set of actions, T and R are respectively the transition probabilities and rewards that are functions of the starting state, the ending state and the chosen action. The most used optimization criterion consists in maximizing the infinite horizon sum $E\left(\sum_{t=0}^{\infty} \beta\, r_t\right)$ 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 probability of a system failure (mission end) between two time points.

The optimization of MDPs produces a *policy*, i.e. a map associating an optimal action to each possible state. It 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 if we start 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 π is solution of the Bellman equation (Bellman 1957) :

$$V^{\pi}(s) = \sum_{s' \in S} T(s, \pi(s), s') \cdot (R(s, \pi(s), s') + \beta\, V^{\pi}(s'))$$

In the probabilistic track of IPC'06 (Bonet & Givan 2005), no rewards are explicitly associated to transitions between states. Alternatively, some goal states are defined, meaning that the planner must produce a policy that aims at reaching at least one goal state from some possible initial states. In an MDP framework, this approach is equivalent to define positive rewards to the transitions that lead to goal states. As all goal states have the same significance, all rewards have the same value.

In this particular case, the policy that maximizes the accumulated expected rewards is equal to the policy that maximizes the probability of reaching the goal state subspace $\mathcal{G}$. For a given policy π, the probability P^{π} to reach at least one goal state from any state convergences and it satisfies the following probabilistic dynamic programming equation (Teichteil-Königsbuch 2005):

$$P^{\pi}(s) = \mathbf{1}_{\mathcal{G}}(s) + \mathbf{1}_{\mathcal{S} \setminus \mathcal{G}}(s) \cdot \sum_{s' \in \mathcal{S}} T(s, \pi(s), s') \cdot P^{\pi}(s')$$

where $\mathbf{1}_{\mathcal{E}}$ is the indicator function of a subspace $\mathcal{E} \subset \mathcal{S}$.

State space factorization

Our planner uses a compact factored representation of MDPs based on Algebraic Decision Diagrams (ADDs) (R.I. Bahar *et al.* 1993) that generalize Binary Decision Diagrams (BDDs). Our model is based on work by (Hoey *et al.* 2000) to model and optimize MDPs with decision diagrams. Since the problems of the stochastic planning track of the competition are given in an extension of the PPDDL 1.0 language

(Younes & Littman 2003), we must translate the PPDDL domain and problem definitions into ADDs-based MDP representation. We used the CUDD package (Somenzi 1998) to deal with ADDs and BDDs in the competition.

The factorization of the state space consists in a cross product involving binary state variables: $\mathcal{S} = \otimes_{i=1}^{n} \mathcal{V}_i$. These variables are the instantiations of the PPDDL parametrized predicates for each constant and each object. It is a compact representation because the states are not enumerated in a list, but rather structured by the set of random state variables. Such variables enable to process sets of states, instead of individual states, whenever useful.

The actions are obtained by instantiating all PPDDL parametrized actions for all constants and objects. For each action, the transition probability function can be represented by a Dynamic Bayesian Network (DBN) (Dean & Kanazawa 1989). It encodes the probabilistic effects and rewards obtained on the different values of 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*).

The factored conditional transition probabilities of the DBNs can be encoded as ADDs, that internally use *unprimed* (pre-action) variables and *primed* (post-action) variables (Hoey *et al.* 2000). The action masks, i.e. PPDDL preconditions, can be encoded as BDDs, since they can be defined as indicator functions.

Dealing with action similarities

In PPDDL, actions often are similar in the sense that some parameter instantiations lead to the same preconditions or effects for two different actions. Therefore, some sub-diagrams of the ADDs encoding the transitions are the same over the different actions. In order to memorize only one time these sub-diagrams, we propose to merge all the transition ADDs (resp. mask BDDs) of each action in a single ADD (resp. BDD) named *Global Action Diagram*. This requires to define action variables shared by all ADDs and BDDs: if PPDDL parametrized actions lead to m instantiated actions for all constants and objects, we must introduce $\mathbb{E}(\log_2 m)$ action binary variables on top of primed and unprimed state variables ($\mathbb{E}(k)$ is the smallest integer bigger or equal to k). Also, our single transition ADD $\tilde{T}$ is defined as:

$$\tilde{T} = \sum_{a \in \mathcal{A}} \mathbf{1}_a \cdot T^a$$

Contrary to work by (Hoey *et al.* 2000), our policy encoding is no more a list of mask BDDs for each action, but rather a single mask BDD that represents the state subspace (unprimed variables) where each action is optimal:

$$\pi = \bigcup_{a \in \mathcal{A}} \mathbf{1}_a \cdot \pi^{-1}(a)$$

In the value iteration scheme, the update of the value requires to compute the maximum of the previous computed value over the actions (Puterman 1994). This can be tedious and ineffective with action variables because the value of an action can only be retrieved by a projection on the variables that encode this action. The value update step would require as many projection computations as the number of actions, and each projection would cause the loss of similarities between actions.

Therefore, we have extended the CUDD package by a new low-level ADD function that we called `Cudd_addMaximumAbstract`, and inspired by `Cudd_addExistAbstract`. This new function computes the maximum of all sub-diagrams over the variables of a given cube. In the case of MDPs, this cube is composed of the action variables. It recursively calls the built-in CUDD function `Cudd_addMaximum` on the sub-diagrams of each action variable, until all action variables are parsed.

Optimization focused on the goal states

In the problems of the competition, the knowledge of possible initial states and of goal states enables to restrict the policy computation to a subspace of the entire state space. This idea was already used in `sLAO*`, but it only used the knowledge of initial states. Moreover, `sLAO*` is based on a heuristic that is an approximation of the value of states over the entire state space, that then helps the optimal optimization of the value on the states that are reachable from the initial states. As a consequence, `sLAO*` requires to initially visit all the states in the heuristic computation step.

Deterministic reachability analysis

Therefore, we propose to compute a subset of reachable states *before* any approximate or optimal dynamic programming computation, by using the knowledge of *both* initial and goal states. This initial step is performed by making all transitions deterministic: in other words, we transform the Global Action Diagram ADD into a BDD by replacing all non-zero discriminants by 1. As a result, we can efficiently propagate the fringe of reachable states from the initial states until at least one goal state is reached, without memorizing the actions that lead from the initial states to the goal states. Moreover, it is known that BDDs are more effective than 1-0 ADDs (R.I. Bahar *et al.* 1993). This forward reachable state search satisfies the following recursive equation:

$$\mathbf{1}_{\mathcal{F}'^{t+1}}(s') = \bigcup_{a \in \mathcal{A}} \bigcup_{s \in \mathcal{S}} \tilde{T}^{det}(s,a,s') \cdot \mathbf{1}_{\mathcal{F}^t}(s)$$

where $\mathcal{F}$ is the current subset of (forward) reachable states and $\tilde{T}^{det}$ is the Global Action Diagram BDD.

This reachable state subset can still be reduced by performing a backtrack search of reachable states from goal states to initial states inside the forward reachable state subset:

$$\mathbf{1}_{\mathcal{B}^{t-1}}(s) = \bigcup_{a \in \mathcal{A}} \bigcup_{s' \in \mathcal{S}} \tilde{T}^{det}(s,a,s') \cdot \mathbf{1}_{\mathcal{F}}(s) \cdot \mathbf{1}_{\mathcal{F}'}(s') \cdot \mathbf{1}_{\mathcal{B}'^t}(s')$$

where $\mathcal{B}$ is the current subset of (backward) reachable states.

Safest stochastic path policy

After we have generated the initial reachable state subspace $\mathcal{W}$ ($= \mathcal{B}$ at the end of the backward deterministic reachability analysis), we compute the policy that maximizes the probability of reaching the goal state subspace $\mathcal{G}$ inside $\mathcal{W}$. We named this policy *safest stochastic path policy*, since it maximizes the chance of reaching at least one goal state. In the probabilistic track of IPC'06, there are no (positive or negative) rewards so that this policy is the same as the classical optimal policy of MDPs obtained if each goal state is rewarded by 1.

In this special case, the maximum probability of reaching the goal state subspace always converges (Teichteil-Königsbuch 2005). Therefore, contrary to the computation of the value function in MDPs, this probability does not require to be pondered by an empirical discount factor at each iteration (Puterman 1994). The maximum probability P of reaching $\mathcal{G}$ inside $\mathcal{W}$ is given by:

$$P^{t-1}(s) = \mathbf{1}_{\mathcal{G}}(s) + \mathbf{1}_{\mathcal{W}\setminus\mathcal{G}}(s) \cdot \max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} T(s,a,s') \cdot \mathbf{1}_{\mathcal{W}}(s) \cdot \mathbf{1}_{\mathcal{W}'}(s') \cdot P^t(s')$$

The maximum over the actions is performed by the function `Cudd_addMaximumAbstract` that we added to CUDD (Somenzi 1998) as an extension.

Policy refinement

The previously obtained policy is not guaranteed to be optimal over the entire state space since it is only optimized over $\mathcal{W} \subset \mathcal{S}$. In order to improve the policy, we alternate a step of deterministic reachability analysis and a step of safest stochastic path policy optimization, in a loop that ends when the policy convergences over the previous reachable state subspace. The current reachable state subspace is generated with the same function as the one responsible for the initial computation of the reachable states, but the fringe of state expansion is propagated by following the current policy (and not by applying all possible actions).

The forward reachable state subspace expansion is:

$$\mathbf{1}_{\mathcal{F}'^{t+1}}(s') = \bigcup_{a\in\mathcal{A}} \bigcup_{s\in\mathcal{S}} \tilde{T}^{det}(s,a,s') \cdot \pi(a,s) \cdot \mathbf{1}_{\mathcal{F}^t}(s)$$

where π is a BDD depending on action variables and on unprimed state variables. The backward expansion is given by:

$$\mathbf{1}_{\mathcal{B}^{t-1}}(s) = \bigcup_{a\in\mathcal{A}} \bigcup_{s'\in\mathcal{S}} \tilde{T}^{det}(s,a,s') \cdot \pi(a,s) \cdot \mathbf{1}_{\mathcal{F}}(s) \cdot \mathbf{1}_{\mathcal{F}'}(s') \cdot \mathbf{1}_{\mathcal{B}'^t}(s')$$

Related work

`sLAO*` (Feng & Hansen 2002) already uses such alternation of a step of reachable state expansion by following the current policy and of a step of policy optimization over the current reachable state subspace. However, there are some key differences between `sLAO*` and our algorithm. First, `sLAO*` does not use the knowledge of goal states so that the state space expansion stops as soon as the new reachable states are the same as the previous reachable states in the high-level loop. Second, for the same reason, the state space expansion step of `sLAO*` does not perform a backward search of reachable states. Third, `sLAO*` takes into account general (positive or negative) rewards so that the policy is updated with the classical discounted dynamic programming equation of MDPs. Nevertheless, the latter is not really a drawback of our algorithm since we can replace the safest stochastic path policy by a classical MDPs policy without changing the spirit of our framework. Moreover, in case of general rewards, the safest stochastic path policy can be a heuristic to the policy optimized with rewards (Teichteil-Königsbuch 2005).

Conclusion

We have presented `sfDP` (*Symbolic Focused Dynamic Programming*) that participates to the probabilistic track of the 2006 International Planning Competition. Based on Binary and Algebraic Decision Diagrams, it is a symbolic dynamic programming algorithm for planning under action uncertainty that focuses the policy on goal states. A step of reachable state subspace expansion by following the current policy alternates with a step of policy optimization over the current reachable state subspace, in a loop that ends if the policy stabilizes over the previous reachable state subspace. We think that the competition is a good forum to improve our algorithm, to compare it with other approaches, and to exchange interesting ideas between researchers.

References

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

Bonet, B., and Givan, R. 2005. 5th international planning competition: Non-deterministic track call for participation.

Dean, T., and Kanazawa, K. 1989a model for reasoning about persistence and causation*Computational Intelligence* 5(3): 142–150.

Feng, Z., and Hansen, E. 2002. Symbolic heuristic search for factored markov decision processes. In *Proceedings 18th AAAI*, 455–460.

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.

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.

Somenzi, F. 1998. Cudd: Cu decision diagram package release.

Teichteil-Königsbuch, F. 2005. *Symbolic and Heuristic Approach of Planning under Uncertainty*. Ph.D. Dissertation, SUPAERO.

Younes, H. L., and Littman, M. L. 2003. PPDDL 1.0: An extension to PDDL for expressing planning domains with probabilistic effects.