Propositional Planning

Michael Paul Fourman
Institute for Representation and Reasoning, and
Laboratory for Foundations of Computer Science,
Informatics, The University of Edinburgh, Scotland, UK.
`Michael.Fourman@ed.ac.uk`

Abstract

We describe a general setting for finite-state planning, where planning operators, or actions, act as functions mapping sets of states to sets of states. In particular, we introduce *propositional actions*, which generalise STRIPS actions. Sets of states may be represented by Binary Decision Diagrams (BDDs), and propositional actions may be directly encoded as efficient operations on BDDs. We describe PropPlan a planner based on this representation, report its performance on a variety of benchmark problems, and discuss its relation to other BDD-based approaches to planning, and to GraphPlan, by which it was inspired.

Planning was historically treated operationally by exploring either the state space or the plan space. However, both these spaces grow exponentially with problem size, and naïve search is not feasible for all but the smallest examples. GraphPlan (Blum & Furst 1997) introduced a new data structure, the planning graph. This planning graph provides a global analysis of the planning problem. From the planning graph we can see that that, at a given stage in the planning process, some states cannot be reached, and some sets of actions cannot be concurrently executed. Using this information allows the search for a plan to be pruned, often drastically. The GraphPlan analysis is approximate: the planning graph does not, in general, exclude all unreachable states or all incompatible sets of actions. For some domains, GraphPlan is unable to exclude any states after the first few steps of exploration.

We present an algorithm, PropPlan , inspired by GraphPlan . Our algorithm is a recasting of naïve, breadth-first state-space exploration. However, rather than explore the state space concretely, by generating and visiting individual states, we explore it abstractly, by computing reachable *sets* of states. PropPlan , like PropPlan , first builds a layered data structure, in our case recording, exactly, the sets $\mathcal{Y}_i$ of states reachable in i steps. PropPlan then uses this information to build plans.

GraphPlan , builds plans with possibly several, non-interfering, parallel actions at each step, a form of partially-ordered plan. Our algorithm produces totally-ordered, sequential plans, executing a single action at each step.

PropPlan , represents sets of states using ordered Binary Decision Diagrams (OBDDs) (Bryant 1986). This makes *exact* representation of the sets of reachable states tractable for a variety of standard examples.

Domains, situations and plans

Definition 1 (domains, states and actions) *A planning* domain, $\mathbb{D}$ *consists of*

- *a set* $\mathbf{S}_{\mathbb{D}}$ *of* states,
- *a set* $\mathbf{A}_{\mathbb{D}}$ *of* actions, *and*
- *for each action* $\mathsf{A} \in \mathbf{A}$ *a transition relation*

$$\cdot \xrightarrow[\mathbb{D}]{\mathsf{A}} \cdot \quad \subseteq \quad \mathbf{S} \times \mathbf{S}$$

The state set and action relations of a planning domain form a labelled transition system, a directed graph with edges labelled by actions.

For $\mathcal{X}, \mathcal{Y} \subseteq \mathbf{S}$ sets of states, and A an action, we write

$$\mathcal{X} \triangleright_{\mathsf{A}} \mathcal{Y} \text{ for } \{y \in \mathcal{Y} \mid \exists x \in X\ x \xrightarrow{\mathsf{A}} y\} \tag{1}$$

$$\mathcal{X} \triangleleft_{\mathsf{A}} \mathcal{Y} \text{ for } \{x \in \mathcal{X} \mid \exists y \in Y\ x \xrightarrow{\mathsf{A}} y\} \tag{2}$$

Definition 2 (situations) *A planning* situation, $\mathbb{S}$*, is given by a domain,* $\mathbb{D}_{\mathbb{S}}$*, together with two distinguished state sets*[1]*:*

$$\mathtt{init}_{\mathbb{S}} \quad \subseteq \quad \mathtt{constraint}_{\mathbb{S}} \quad \subseteq \quad \mathbf{S}_{\mathbb{D}}$$

We normally omit the domain and situation subscripts, $(-)_{\mathbb{D}}, (-)_{\mathbb{S}}$, where these can be inferred from the context.

[1]The set `constraint` is used to represent a global constraint on the states that may be visited by a plan, for example, to implement domain axioms. The reader will not be misled if she assumes, for the time being, that $\mathtt{constraint}_{\mathbb{S}} = \mathbf{S}_{\mathbb{D}}$.

Given a situation, $\mathbb{S}$, each action gives two functions from sets of states to sets of states—corresponding to applying the action either forward or backward. If $\mathcal{X} \subseteq \mathbf{S}$ is a set of states, and A an action, we write

$$\overrightarrow{\mathsf{A}}(\mathcal{X}) = \mathcal{X} \rhd_{\mathsf{A}} \texttt{constraint} \tag{3}$$

$$\overleftarrow{\mathsf{A}}(\mathcal{Y}) = \texttt{constraint} \lhd_{\mathsf{A}} \mathcal{Y} \tag{4}$$

Given a sequence $\langle \mathsf{A}_i \rangle$ of actions, we define $\overrightarrow{\langle \mathsf{A}_i \rangle}$ to be the composition of the functions $\overrightarrow{\mathsf{A}_i} : \mathcal{P}(\mathbf{S}) \to \mathcal{P}(\mathbf{S})$.

The set $\overrightarrow{\mathsf{A}}(\mathcal{X})$ represents the states reachable by applying A to states in $\mathcal{X}$, while $\overleftarrow{\mathsf{A}}(\mathcal{Y})$ is the set of states from which $\mathcal{Y}$ is reachable by applying A.

Definition 3 (goals, plans and achievement) *A* goal *is a set of states; a* plan *is a sequence of actions.*

Given a situation, $\mathbb{S}$, we say the plan, $\langle \mathsf{A}_i \rangle$, achieves *a goal, $\mathcal{G}$, iff*

$$\overrightarrow{\langle \mathsf{A}_i \rangle}(\texttt{init}_{\mathbb{S}}) \textit{ intersects } \mathcal{G}.$$

We also say a plan achieves *a state, y, iff it achieves the singleton set $\{y\}$, and that a set, $\mathfrak{P}$, of plans* covers *a goal, $\mathcal{G}$, iff each $y \in \mathcal{G}$ is achieved by some plan in $\mathfrak{P}$.*

So, if we take a situation and a goal, and look for plans that achieve the goal, the *problem* we set ourselves is to produce a sequence of actions that will take us from *some* state in the initial set to *some* state in the goal set, without leaving the constraint set.

If P is a plan and A an action, we write $P\frown\mathsf{A}$ for the plan obtained by appending A to P, and if $\mathfrak{P}$ is a set of plans, we write $\mathfrak{P}\frown\mathsf{A}$ for $\{P\frown\mathsf{A} \mid P \in \mathfrak{P}\}$

The PROPPLAN algorithm

Our planning algorithm takes a situation and a goal, and solves the problem of finding plans that achieve the goal. It has two stages: first (step 1) we **build** a data structure, then (step 2) we **plan**.

Step 1: build

We build a layering of the state space. This data structure plays a rôle analogous to that of the planning graph in GraphPlan.

- Iteratively create the sequence $\langle \mathcal{X}_i \rangle$ of sets of states reachable from `init` via a path containing at most i actions.

$$\mathcal{X}_0 = \texttt{init}$$
$$\mathcal{X}_{i+1} = \bigcup_{\mathsf{A} \in \mathbf{A}} \overrightarrow{\mathsf{A}}(\mathcal{X}_i)$$

  Terminate whenever, either, the goal is achieved, ($\texttt{reached} = \mathcal{X}_i \cap \texttt{goal}$ is non-empty), or no new states are produced ($\mathcal{X}_{i+1} = \mathcal{X}_i$). In the latter case there is no solution to this planning problem; terminate, returning no plans.

  Since the state space is finite, and each $\overrightarrow{\mathsf{A}}$ is monotone, one of these cases must eventually occur.

- While building the sequence $\langle \mathcal{X}_i \rangle$, create the sequence, $\langle \mathcal{Y}_i \rangle$, of set differences:

$$\mathcal{Y}_0 = \mathcal{X}_0 \qquad \mathcal{Y}_{i+1} = \mathcal{X}_{i+1} \setminus \mathcal{X}_i$$

  The set $\mathcal{Y}_i$ consists of states reachable in i, but no fewer, steps from the initial state.

- Also construct, for each step, i, the set, $\mathcal{A}_i$, of those actions, A, such that $\overrightarrow{\mathsf{A}}(\mathcal{Y}_i)$ contributes newly reachable states to $\mathcal{Y}_{i+1}$

If the goal *is* reachable in n steps, then **build** terminates after step n, with a non-empty set of reachable goal states:

$$\texttt{reached} = \mathcal{X}_n \cap \texttt{goal} = \mathcal{Y}_n \cap \texttt{goal} \tag{5}$$

Lemma 1 *Let $\langle \mathcal{Y}_i \rangle_{i \le n}$ and $\langle \mathcal{A}_i \rangle_{i<n}$ be constructed as above. For all $m \le n$, and for all $y \in \mathcal{Y}_m$, there is a plan achieving y.*

Proof By induction on m.

For $m = 0$ we have $y \in \texttt{init}$ — the empty plan, $\langle\rangle$, achieves all goals. Now suppose $m > 0$ and $y \in \mathcal{Y}_m$. By construction,

$$\mathcal{Y}_m \subseteq \bigcup_{\mathsf{A} \in \mathcal{A}_{m-1}} \overrightarrow{\mathsf{A}}(\mathcal{Y}_{m-1})$$

So, $y \in \overrightarrow{\mathsf{A}}(\mathcal{Y}_{m-1})$ for some $\mathsf{A} \in \mathcal{A}_{m-1}$. By definition of $\overrightarrow{\mathsf{A}}$, there exists $x \in \mathcal{Y}_{m-1}$ such that $x \xrightarrow{\mathsf{A}} y$.

By the induction hypothesis, there is a plan, P, achieving x; the plan $P\frown\mathsf{A}$ achieves y.

■

Before describing our algorithm for constructing plans from the layered state space, we prove that plans exist to cover any *set* of reachable goals — a variant of Lemma 1. This is of course an immediate corollary, but it is worthwhile going through a separate proof, an induction based on manipulating sets rather than individual states, as our algorithm can then be read directly from this proof.

Lemma 2 *Let $\langle \mathcal{Y}_i \rangle_{i \le n}$ and $\langle \mathcal{A}_i \rangle_{i<n}$ be constructed as above. For all $m \le n$, and for all $\mathcal{G} \subseteq \mathcal{Y}_m$, there is a set of plans covering $\mathcal{G}$.*

Proof By induction on m.

For $m = 0$ we have $\mathcal{G} \subseteq \texttt{init}$ — the empty plan is the only plan we need. Now suppose $m > 0$ and $\mathcal{G} \subseteq \mathcal{Y}_m$. By construction,

$$\mathcal{Y}_m \subseteq \bigcup_{\mathsf{A} \in \mathcal{A}_{m-1}} \overrightarrow{\mathsf{A}}(\mathcal{Y}_{m-1})$$

so it suffices to show, for each $\mathsf{A} \in \mathcal{A}_{m-1}$, that there exists a set of plans covering

$$\mathcal{Y}_{m-1} \rhd_{\mathsf{A}} \mathcal{G} = \mathcal{G} \cap \overrightarrow{\mathsf{A}}(\mathcal{Y}_{m-1})$$

By induction hypothesis, there is a set, $\mathfrak{P}$, of plans covering $\mathcal{Y}_{m-1} \lhd_{\mathsf{A}} \mathcal{G} \subseteq \mathcal{Y}_{m-1}$. But,

$$\mathcal{Y}_{m-1} \lhd_{\mathsf{A}} \mathcal{G} = \left\{ x \in \mathcal{Y}_{m-1} \;\middle|\; \exists y \in \mathcal{G}\; x \xrightarrow{\mathsf{A}} y \right\} \qquad (6)$$

so any plan achieving $\mathcal{Y}_{m-1} \lhd_{\mathsf{A}} \mathcal{G}$ can be extended, by A, to a plan achieving $\mathcal{G}$. In passing, we note, for future reference, that *all* minimal-length plans achieving $\mathcal{G}$, whose final action is A, arise in this way.

Finally, observe that,

$$\forall y \in \mathcal{G}_{\mathsf{A}}\; \exists x \in (\mathcal{Y}_{m-1} \lhd_{\mathsf{A}} \mathcal{G})\; x \xrightarrow{\mathsf{A}} y$$

So the set, $\mathfrak{P}^\frown\mathsf{A}$, of plans obtained by appending A to each plan in $\mathfrak{P}$, covers $\mathcal{G}_{\mathsf{A}}$.

■

Step 2: plan

This part of the algorithm is based on the proof of Lemma 2.

Define a recursive function, `plan`, which takes as arguments an integer, m, and a non-empty set, $\mathcal{G} \subseteq \mathcal{Y}_m$, of reachable goals, and returns a set of plans covering $\mathcal{G}$, as follows:

$$\begin{aligned} \texttt{plan}(0, \mathcal{G}) &= \{\langle\rangle\} \\ \texttt{plan}(m, \mathcal{G}) &= \bigcup_{\mathsf{A} \in \mathcal{A}_{m-1}} \texttt{plan}(m-1, \mathcal{Y}_{m-1} \lhd_{\mathsf{A}} \mathcal{G})^\frown\mathsf{A} \end{aligned}$$

Now apply `plan` to the reachable goals found earlier (equation 5), and return the set

$$\texttt{plan}(n, \texttt{reached})$$

We implement sets of actions, sets of plans, and sets of sets of plans, as lazy lists, so that we only actually compute the plans we need.

Theorem 1 PropPlan *returns the set of all plans of minimal length that lead from some initial state to some goal state.*

Proof By induction on m and Lemma 2, the call `plan` $(m, \mathcal{G})$, where $\mathcal{G} \subseteq \mathcal{Y}_m$, returns the set of all minimal-length plans achieving $\mathcal{G}$. By Lemma 1, these cover $\mathcal{G}$. Finally, by construction, `reached` is precisely the set of goalstates achieved by some minimal-length plan. ■

Representing sets of states

We now specialise to the (standard) situation where states are determined by the values of a finite number of Boolean state variables. A valuation, $\mathbf{v}$, is a function assigning a Boolean value, $\mathbf{v}(\dot{x})$, to each state variable, $\dot{x}$. We write $\mathbf{v}[b/x]$ for the valuation $\mathbf{w}$ such that $\mathbf{w}(\dot{x}) = b$ and $\mathbf{w}(y) = \mathbf{v}(y)$ for $y \neq \dot{x}$. Each state corresponds to a valuation on the variables; for n variables we have 2^n states. A Boolean function in the state variables is a Boolean-valued function, $\varphi : \mathbf{S} \longrightarrow \{\mathtt{0}, \mathtt{1}\}$, whose domain is the set of all states.

We use the standard notation for Boolean operations on Boolean functions:

$$\neg, \wedge, \vee, \mathtt{0}, \mathtt{1}, \text{ and, for finite conjunctions, } \bigwedge$$

(Here, 0 and 1 denote the constant functions yielding the Boolean values 0 and 1, $\neg$ denotes the function that composes its argument, a Boolean function, with Boolean negation, and so on...)

For each state variable, $\dot{x}$, there is a corresponding Boolean function, x, the projection of the state space onto that variable: $x(\mathbf{v}) = \mathbf{v}(\dot{x})$. We may also treat each valuation as a Boolean function: $\mathbf{v}(\mathbf{v}) = \mathtt{1}$, and $\mathbf{v}(\mathbf{w}) = \mathtt{0}$ for $\mathbf{w} \neq \mathbf{v}$.

We write $\varphi[\psi/x]$ for the result of substituting ψ for x in φ.

$$\varphi[\psi/x](\mathbf{v}) = \varphi(\mathbf{v}[\psi(\mathbf{v})/x])$$

If x is a variable, we write

$$\exists x\; \varphi \text{ for the function } \varphi[\mathtt{0}/x] \vee \varphi[\mathtt{1}/x]$$

We can quantify similarly over a set of variables. The *support* of φ is the set of variables, x, such that $\varphi \neq \exists x\; \varphi$ (equivalently, such that $\varphi \neq \varphi[\mathtt{0}/x]$, etc. ...).

Ordered Binary Decision Diagrams (OBDDs) (Bryant 1986), provide a canonical, and often effective, representation of Boolean functions of a finite set of variables. They support efficient (i.e. linear in the sizes of the diagrams involved) implementations of the Boolean operations, of Boolean quantification, and of the function `support`. BDDs have been extensively used in system verification to provide a tractable tool for exploring large, but finite, state spaces (Clarke, Grumberg, & Peled 1999).

PropPlan uses BDDs to represent sets of states. Our algorithm uses Boolean operations on sets of states, together with the functions $\rhd_{\mathsf{A}}$ and $\lhd_{\mathsf{A}}$ associated with an action. To implement the algorithm, it therefore remains to describe the implementation of these functions as operations on BDDs. We do this first for STRIPS operators.

Propositional planning

In this section we discuss the representation of planning operators as operations on Boolean functions. We begin by showing how STRIPS operators are encoded in PropPlan , and then generalise.

STRIPS planning

STRIPS planning was introduced by (Fikes & Nilsson 1971). In this setting, we have a finite set P of properties, generated by applying a finite set of predicates to a finite set of objects. States are finite sets of properties. Actions are generated by instantiating a finite number of parametrised axiom schemas.

Each STRIPS action, A, is characterised by three sets of properties:

$$\texttt{precond}_{\mathsf{A}}, \quad \texttt{add}_{\mathsf{A}}, \quad \texttt{remove}_{\mathsf{A}}$$

STRIPS operators correspond to partial functions from states to states.

$$\text{If } \mathtt{precond}_{\mathsf{A}} \subseteq x \text{ then } x \xrightarrow{\mathsf{A}} (x \cup \mathtt{add}_{\mathsf{A}} \setminus \mathtt{remove}_{\mathsf{A}})$$

Any variables not explicitly mentioned in the effects of an action are unchanged by it (the *frame assumption*).

We use an extract from the `blocksworld` domain from (McDermott 1999), which is expressed in PDDL notation (McDermott *et al.* 1998) notation, to illustrate our constructions. Here is an action schema from the `blocksworld` domain:

```
(:action stack
 :parameters  (?ob ?underob)
 :precondition (and (clear ?underob)
                    (holding ?ob))
 :effect (and (arm-empty)
              (clear ?ob)
              (on ?ob ?underob)
              (not (clear ?underob))
              (not (holding ?ob))))
```

This is a PDDL representation of a STRIPS action schema. It can be instantiated for every pair of blocks. For example, if a and b are blocks, we instantiate `stack(?ob ?underob)` as $\mathtt{stack}(a,\ b)$ to produce an action, A, with the following STRIPS representation:

$$\begin{aligned} \mathtt{precond}_A &= \{(\mathtt{clear}\ b), (\mathtt{holding}\ a)\} \\ \mathtt{add}_A &= \{(\mathtt{arm\text{-}empty}), (\mathtt{clear}\ a), (\mathtt{on}\ a\ b)\} \\ \mathtt{remove}_A &= \{(\mathtt{clear}\ b), (\mathtt{holding}\ a)\} \end{aligned}$$

To place STRIPS planning in our propositional setting, we take each property $p \in \mathsf{P}$ as a state variable. In our representation, the STRIPS action, A, is characterised by

$$\begin{aligned} \mathtt{precondition} &= \bigwedge_{p \in \mathtt{precond}_{\mathsf{A}}} p \\ \mathtt{effect} &= \bigwedge_{a \in \mathtt{add}_{\mathsf{A}}} a \wedge \bigwedge_{r \in \mathtt{remove}_{\mathsf{A}}} \neg r \\ \mathtt{changes} &= \mathit{support}(\mathtt{effect}) \\ &= \mathtt{add}_{\mathsf{A}} \cup \mathtt{remove}_{\mathsf{A}} \end{aligned}$$

So, returning to our example, $\mathtt{stack}(a,\ b)$ is represented by

$$\begin{aligned} \mathtt{precondition} &= (\mathtt{clear}\ b) \wedge (\mathtt{holding}\ a) \\ \mathtt{effect} &= (\mathtt{arm\text{-}empty}) \wedge (\mathtt{clear}\ a) \wedge (\mathtt{on}\ a\ b) \\ &\quad \wedge \neg(\mathtt{clear}\ b) \wedge \neg(\mathtt{holding}\ a) \\ \mathtt{changes} &= \{(\mathtt{arm\text{-}empty}), (\mathtt{clear}\ a), \\ &\quad (\mathtt{on}\ a\ b), (\mathtt{clear}\ b), (\mathtt{holding}\ a)\} \end{aligned}$$

Now we turn to the propositional representation of $\lhd_{\mathsf{A}}$ and $\rhd_{\mathsf{A}}$. Consider $\mathcal{X} \rhd_{\mathsf{A}} \mathcal{Y}$; we have to determine the states in $\mathcal{X}$ for which A is enabled; modify those states to produce the results of applying A, and intersect the result with $\mathcal{Y}$. If the BDDs $\mathcal{X}$ and $\mathcal{Y}$ represent sets of states then $\mathcal{X} \rhd_{\mathsf{A}} \mathcal{Y}$ is represented by the BDD

$$\exists \mathtt{changes}(\mathcal{X} \wedge \mathtt{precondition}) \wedge \mathtt{effect} \wedge \mathcal{Y}$$

First we conjoin $\mathcal{X}$ with `precondition` to determine the states in which that action is enabled. Then we use existential quantification to discard the current values of the changing variables, and conjoin with `effect` to set the new values of these variables. Finally, we conjoin with $\mathcal{Y}$.

Similarly, and symmetrically, $\mathcal{X} \lhd_{\mathsf{A}} \mathcal{Y}$ is represented by the BDD

$$\mathcal{X} \wedge \mathtt{precondition} \wedge \exists \mathtt{changes}(\mathtt{effect} \wedge \mathcal{Y})$$

Propositional actions

PDDL accomodates a number of generalisations of the STRIPS formalism. In particular, an action's `precondition` is a logical formula, which may include arbitrary boolean connectives, equality, and quantification over a finite domain of objects. PDDL requires an action's `effect` to be a conjunction of literals (atoms or their negations).

Our representation is more general. Each propositional action is characterised by two propositions: its `precondition` and its `effect`; and a set, `changes`, of variables, these are the variables *not* subject to the *frame assumption*—the variables that may be changed by the action A.

This representation is simply based on what we need, in order to define $\rhd_{\mathsf{A}}$ and $\lhd_{\mathsf{A}}$ as above. It extends the STRIPS notion of action in various ways.

Generalised preconditions First, in common with various other formalisms, including PDDL, we allow arbitrary Boolean functions as preconditions. Quantifications over finite domains are expanded to conjunctions or dijunctions.

Generalised effects Second, we allow allow arbitrary Boolean functions as effects. The implicit operational semantics for these is a form of benign choice. The Boolean function represents the set of possible effects of the action; we accept a plan if it will achieve our goal provided we make the right choices.

Limited changes Third, the frame-connection between one state and the next is expressed in our setting by only allowing each action to change a limited set of variables. Making this set independent of the formula specifying the action's effect allows two possibilities.

By making the set of changeable variables larger, we can relax the frame assumption and consider plans that at some steps call on benign choice to generate appropriate values for particular variables. A similar feature is included in the language $\mathcal{AR}$ of (Cimatti *et al.* 1997).

Perhaps more usefully, we can allow the effect of an action to relate variables the action can change to ones it cannot. In the sequential setting this is merely a notational convenience. If we allow for concurrent execution of a number of actions, it allows us to express some forms of concurrent, cooperative behaviour, that would normally require sequential execution. For example, *Put x into an unoccupied box*, can be formalised

Problem	Problem size	Plan length	Time
gripper	10	65	76.2
gripper	20	125	1029.0
blocksworld	8	14	182.2
blocksworld	10	18	3019.3
hanoi	6	63	50.4
hanoi	8	255	531.0
fixit	—	19	1.0

Figure 1: Timings in seconds

so that it is possible to concurrently execute a number of instantiations (provided there are enough empty boxes).

On the negative side, an ADL action (Pednault 1989) with conditional effects cannot be implemented directly by a single propositional action—although it is straightforward to compile an ADL action with n conditional effects to 2^n propositional actions, each corresponding to one of the 2^n subsets of the n conditional effects.

The semantics of constraints

Our declarative treatment of the planning problem gives a non-directed semantics to arbitrary domain axioms. A constraint on the states a plan may visit simply restricts the set of possible effects of an action. So, if $p \wedge q \rightarrow r$ is a constraint, then an action whose effect implies $\neg r$ can only produce states in which either p or q fails. This may block the action even when its preconditions are satisfied: the set $\overrightarrow{\mathsf{A}}(\mathcal{X})$ may be empty, even when $\mathcal{X} \cap \mathtt{precondition}_{\mathsf{A}}$ is not.

Implementation

Our prototype implementation of PropPlan is written in Standard ML. We use the Poly/ML implementation of SML (Matthews 2000), and, via the PolyML C-interface, access a standard OBDD package (Long 1993), compiled as a shared library. PropPlan parses PDDL files (McDermott *et al.* 1998), and currently handles the following PDDL requirements: `strips`, `typing`, `disjunctive-preconditions`, `equality`. We plan to add `quantified-preconditions`, `conditional-effects`, `domain-axioms`, `open-world`, `true-negation`.

Results and discussion

We have run the prototype version of PropPlan on a variety of examples taken from the GraphPlan home page (Blum, Furst, & Langford 1999), from McDermott's planning problem repository 1999 and from the AIPS-98 planning competition (AIPS98 1999).

Figure 1 tabulates some results from these preliminary experiments (PropPlan version 0.2, running on an Intel i686 under Linux). We report the time taken to for PropPlan produce the first plan.

Raw timings, such as these, are hard to interpret. We now consider, for various planners applied to some

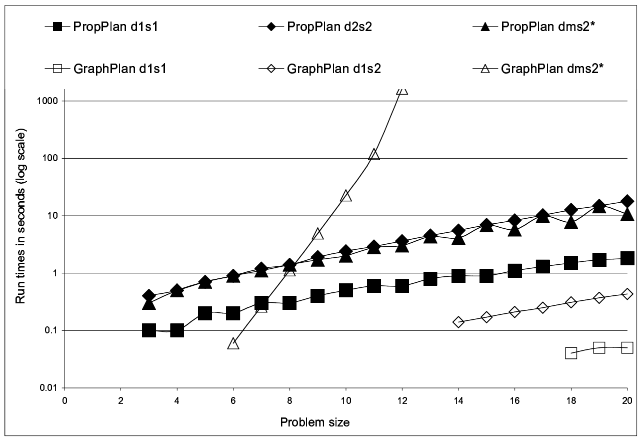

Figure 2: Barrett-Weld domains

standard problems, how the time, t, taken to produce a solution (for PropPlan this is the time taken to produce its first plan) varies with problem size, s. We plot all timings on a log scale to show how $\log t$ varies with s. Typically, for a chosen planner and domain, the timings reported from the competition, plotted for various sizes of problem, fall in roughly a straight line, indicating exponential scaling. Variations in hardware and coding may increase performance by a constant factor, displacing this line vertically; fundamental differences in performance of the underlying algorithms on different problems are reflected in differences between the slopes of the lines for different problems and planners, and in the deviations from linearity: all other things being equal, an algorithm is better if $\partial(\log t)/\partial s$ is smaller, and if $\partial^2(\log t)/\partial s^2$ is more negative.

In Figure 2, we plot timings for a variety of problem sizes of various domains from (Barrett & Weld 1994), for PropPlan and GraphPlan , both running on the same hardware. PropPlan 's performance on `d1s1` and `d1s2` is not impressive. PropPlan takes 31s to solve `d1s2-20`; on the same hardware GraphPlan takes 0.43s to solve this problem. However, for `dms2*`, we see from the graph that PropPlan outperforms GraphPlan for problems of size > 8.

Figure 3 plots our timings for PropPlan on the problems from the `gripper` domain from (AIPS98 1999), together with the timings reported for those planners that solved these problems in the competition. Here we use minimal plan length for the x-axis, as a convenient measure of problem size.

PropPlan always finds minimal-length plans. For the `adl-gripper-x-20` problem of the AIPS-98 competition, PropPlan found a 125-step plan in 1029s. In the competition, there were no solutions to this problem in the ADL track; only HSP solved the strips version of this problem, finding a plan of length 165 in 33.2s. On each of the gripper problems, PropPlan finds a shorter solution than HSP; typically the HSP plan is from 30% to 35% longer than the minimal solution. But HSP's

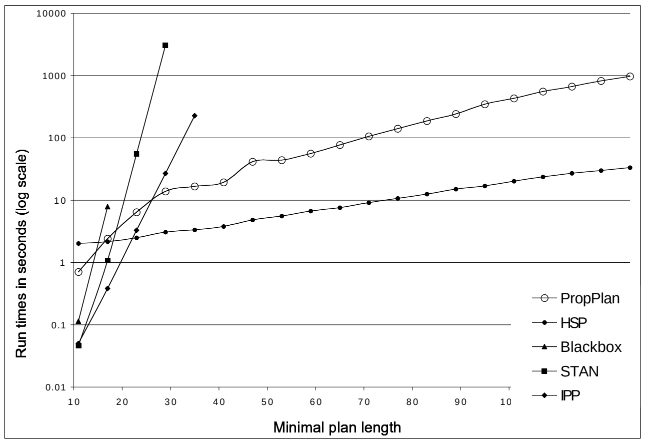

Figure 3: Gripper domain

run times are faster, and HSP's performance scales better with problem size.

HSP is a more traditional planner, that searches the statespace directly. It relies on heuristics to guide the search and invariants to prune the search space. First, it uses an approximation to the domain to compute a lower bound on the minimal plan length from a state to the goal, and uses these estimates as a heuristic. Second, it computes global mutual exclusion invariants, and uses these to prune the search space. It is not clear (to us) why HSP performs so well on the Gripper domain, nor how HSP and PropPlan will compare on other domains.

PropPlan can find *all* minimal-length plans; much of the work is done in finding the first solution. For the `fixit` tyre-changing domain, the standard problem requires a 19-step plan. PropPlan takes 1.0s to find the first solution, and then takes a further 700s to produce all 105,084 minimal solutions to this problem, producing about 150 solutions per second.

Complexity PropPlan 's performance on a particular problem depends linearly on a number of factors: the sizes of the BDDs representing the situation; the sizes of the BDDs representing reachable state sets; the number of instantiated actions, and the number of steps in a minimal plan.

actions The number of *instantiated* actions and predicates is an important factor. Typing can exclude redundant actions and propositions, and this may be critical to the effectiveness of PropPlan . For example, we find the untyped version of the `gripper` problems intractable. Currently, PropPlan instantiates *all* actions and predicates before doing anything else. There is scope for dynamic instantiation: instantiating actions only when their preconditions are found to be reachable; and instantiating propositions only when they become relevant to our representation of the state space. We believe that this will be critical in some domains.

situation The BDDs representing preconditions, effects, initial conditions and goals, for STRIPS problems are just conjunctions of literals; the number of nodes required for each of these is linear in the number of literals.

reachable states In the worst case, the size of BDD required to represent an arbitrary set of states is exponential in the number of state variables, but (and this is true across a wide range of applications of BDDs), in many cases of practical significance this exponential blow-up does not occur. For many functions, the size of BDD used depends critically on the ordering of variables. We have not yet given any thought to specially crafted heuristics for selecting good variable orderings for particularplanning problems or domains. Instead, we have relied on the general-purpose dynamic variable re-ordering heuristics provided by the BDD library, asking this package to reorder the variables after each `build` step, in order to reduce the total number of BDD nodes used.

It is a well-known (Clarke, Grumberg, & Peled 1999) rule-of-thumb that good variable orderings for BDDs place related variables close together in the ordering. Examining the variable orderings found by the heuristics shows that they exploit invariants which relate different variables. For example, mutually exclusive pairs of propositions are often made adjacent in the ordering. More general invariants are also found and exploited. For example, in the gripper domain, a given ball is either in one of the grippers, or in one of the rooms; exactly one of these propositions holds. This functional relationship is exploited by an ordering that makes these propositions adjacent. Here is the variable ordering found by the heuristics for a problem with two grippers, two rooms, and four balls:

```
(%at(ball1 roomb))
(%at(ball1 rooma))
(%carry(ball1 right))
(%carry(ball1 left))
(%free(left))
(%free(right))
(%carry(ball2 right))
(%at(ball2 rooma))
(%at(ball2 roomb))
(%carry(ball2 left))
(%at(ball3 roomb))
(%at(ball3 rooma))
(%carry(ball3 right))
(%carry(ball3 left))
(%at-robby(roomb))
(%at-robby(rooma))
(%at(ball4 rooma))
(%carry(ball4 right))
(%carry(ball4 left))
(%at(ball4 roomb))
```

In all our experiments, we have found that the sizes of the BDDs used to represent the reachable state space tend to grow with each step of the `build` procedure.

We see clear differences in the way these sizes grow for different domains.

For example, in `blocksworld` problems the state sets must encode sets of possible permutations of blocks in a stack. For `blocksworld − tower − 8`, the BDD sizes grow exponentially from step to step for the first 9 steps, as the reachable sets become ever more complex, and then remain roughly unchanged for the remaining five steps. There are 267,886 BDD nodes in use when `build` terminates after 14 steps. For `hanoi` problems this blow-up does not happen; the number of nodes used grows roughly linearly with the number of steps taken. For `hanoi − 8`, after 255 steps, only 62,361 BDD nodes are used. Because of the constraints of this problem, each step introduces very few (always < 256, normally far fewer) newly reachable states.

But the size of the reachable state space is not the issue. For `gripper`, the number of nodes in use grows sub-linearly. For `gripper − 20`, one of the 125 steps introduces 259,978,553,354,520 newly reachable states, but we only use 35,938 BDD nodes to represent the problem and all the layers of reachable states, after the final step of `build`.

Although we have provided for PropPlan search to be limited to states satisfying a given `constraint`, we have not yet experimented to see how imposing constraints may improve performance by simplifying the sets of reachable states, and by excluding some actions from consideration.

The minimal number of plan steps required for a solution is not something we can change. However, we can exploit the fact that our representation of planning is symmetric with respect to the direction of time's arrow. In principle, there is no difference between searching forwards, from `init` to `goal`, and searching backwards, from `goal` to `init`; we simply exchange the preconditions and effects of each action. Since planning gets slower as we take more steps and produce more complex state sets, it will be more efficient to combine these two approaches, building successive layers of states around both `goal` and `init` until the two sets meet in the middle.

In practice, the complexity of the sets of states reachable from different directions varies. (For example, because `goal` is typically less specific than `init`, and because actions designed to model time in its usual direction may, when run backwards from a non-specific goal, introduce bizarre, physically-unreal states.) Since BDD size provides an appropriate measure of complexity for our purposes we can use this to decide which new layer it is most productive to build at any given stage. However, preliminary experiments with this idea are not encouraging. The reordering heuristics no longer find such compact representations of the reachable state sets. This is because the BDDs obtained by stepping backwards from the goal do not share the invariants of those obtained by stepping forward from the initial conditions.

Related work

Planning as model-checking

Guinchiglia & Traverso (1999) describe a closely related approach to planning. A planning problem is represented by a domain (in our sense), together with a goalset, $\mathcal{G}$. In their work, BDDs are used to represent sets of state-actionset pairs. They produce a BDD, representing such a set, $\mathcal{SA}$, of pairs, such that if $(s, \mathbf{a}) \in \mathcal{SA}$ then: first, there is a plan leading from s to $\mathcal{G}$; and, second, $\mathbf{a}$ is a set of actions that can be applied concurrently to s to lead to a state s' nearer to $\mathcal{G}$.

Prompted by the work of Guinchiglia & Traverso, we have experimented with a variant of PropPlan . This version encodes with each reachable state in $\mathcal{Y}_{m+1}$, the action A used to reach it from $\mathcal{Y}_m$. These action tags can be used as a thread of Ariadne, to retrace, from any reachable state, the steps that led us there, we found that this variant built much larger BDDs than the algorithm we have described, and performed some ten-times slower.

One reviewer of an earlier version of this paper drew our attention to the earlier work of Cimatti *et al.* (1997, 1998), which anticipates many aspects of PropPlan . In particular, (Cimatti *et al.* 1997) describes a model-based planner (MBP) with a BDD-based planning algorithm, the first part of which which corresponds directly to our `build` procedure. This is followed by a function, `choose-plan`, closely related to our `plan` function. The difference being that Cimatti *et al.* choose a single state at each step as they retrace their steps to find a plan, whereas we carry a set of states, represented by a BDD(this procedure coresponds to our Lemma 1).

Like GraphPlan, PropPlan builds a layered datastructure encoding information about the set of states reachable after some number of action steps. GraphPlan uses an approximate representation of the set of reachable states. PropPlan uses BDDs to represent exactly the sets of reachable states. Like Blackbox, MBP, and PMC, PropPlan compiles a planning problem to a propositional representation. But, unlike these, PropPlan does not introduce propositional variables to represent actions.

Furthermore, in PropPlan, (generalized) strips operators are represented directly by efficient operations on BDDs in n state variables, rather than, as is traditional in model checking, as abstract transition relations requiring $2n$ variables. PropPlan is directly comparable with PMC, and it will be instructive to compare the sizes of the BDD representations used by the two systems. Thus, the significant innovation we introduce is our representation of plan operators, which appears to confer directly the benefits that might be expected from applying disjunctive partitioning (Clarke, Grumberg, & Peled 1999), to the representation used by both Cimatti *et al.* and Guinchiglia & Traverso.

GRAPHPLAN revisited

GraphPlan uses an approximate representation of sets of properties, and sets of actions.

A set $\mathcal{S}$ of subsets of X is *approximately represented* by a subset $s \subseteq X$, equipped with a symmetric, irreflexive binary *mutex* relation `#`. such that

$$\bigcup \mathcal{S} \subseteq s$$

$$\forall x, y \in X \; x \# y \rightarrow \neg \exists A \in \mathcal{S} \; x \in A \wedge y \in A$$

Planning depends on the construction of a planning graph, a sequence of sets of properties, P_i, and sets of actions, A_i, constructed alternately, starting with $P_0 =$ `initial`. Each of these sets of properties or actions is equipped with a mutex relation, so that the sets of reachable states, and sets of concurrently executable actions, at each stage are approximately represented within the planning graph.

Direct comparison with PropPlan is not possible, as GraphPlan allows, at each stage, for a set of non-interfering actions to be executed concurrently, whereas the PropPlan semantics of actions is strictly sequential. It would be instructive to re-engineer GraphPlan using BDDs to represent, exactly, both the sets of reachable states and the sets of concurrently executable actions.

Planning as satisfiability

(Kautz & Selman 1992) introduced this approach. A planning problem is encoded as a satisfiability problem, by producing formulae P_n such that a plan of length n may be extracted from any valuation satisfying P_n. Radivojević & Brewer (1995, 1993) apply a similar encoding to the scheduling problem for hardware synthesis, and use BDDs to compute valid schedules. These approaches differ from our work, and from that of Guinchiglia & Traverso, in that action-time pairs are treated as propositional variables. Nevertheless, many aspects of the encodings used, and some of the problems encountered, are similar. For example, Kautz & Selman discuss the use use of constraints, similar to those we allow, to simplify the state space in their representation of the planning problem.

References

AIPS98. 1999. Artificial intelligence planning systems competition results. Web page. `http://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html`.

Barrett, A., and Weld, D. 1994. Partial-order planning: evaluating possible efficiency gains. *Artificial Intelligence* 67(1):71–112.

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

Blum, A.; Furst, M.; and Langford, J. 1999. Graphplan home page. Web page. `http://www.cs.cmu.edu ~avrim/graphplan.html`.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* C-35(8):677–691.

Cimatti, A.; Guinciglia, E.; Guinchiglia, F.; and Traverso, P. 1997. Planning via model checking: A decision procedure for AR. In *Proceedings of the Fourth European Conference on Planning – ECP '97.*

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *AAAI/IAAI Proceedings*, 875–881.

Clarke, Jr., E. M.; Grumberg, O.; and Peled, D. A. 1999. *Model Checking.* The MIT Press.

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 5(2):189–208.

Guinchiglia, F., and Traverso, P. 1999. Planning as model checking. In Biundo, S., and Fox, M., eds., *Proceedings of the Fifth European Conference on Planning (ECP99)*, Lecture Notes in Artificial Intelligence, 1–20. Durham, UK: Springer-Verlag.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In Neumann, B., ed., *Proceedings of the 10th European Conference on Artificial Intelligence*, 359–363. John Wiley & Sons.

Long, D. E. 1993. A binary decision diagram (BDD) package. Web page. `http://www.cs.cmu.edu ~modelcheck/bdd.html`.

Matthews, D. 2000. PolyML. www. `http://www.polyml.org`.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL–the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

McDermott, D. 1999. Planning problem repository. `ftp://ftp.cs.yale.edu/pub/mcdermott/domains/`.

Pednault, E. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st Int. Conf. on Principles of Knowledge Representation and Reasoning*, 324–332.

Radivojević, I., and Brewer, F. 1993. Symbolic techniques for optimal scheduling. In *Proc. 4th Synthesis and Simulation Meeting and Int. Interchange (SASIMI).*

Radivojević, I., and Brewer, F. 1995. Symbolic scheduling tehniques. *IEICE Trans. Information and Systems.*