Graphplan Based Conformant Planning with Limited Quantification

Alan Carlin[1], James G. Schmolze[1], Tamara Babaian[2]

[1] Department of Electrical Eng. And Computer Science
Tufts University, Medford, MA 02155 USA
{schmolze, acarli04}@cs.tufts.edu
[2] Department of Computer Information Systems
Bentley College, Waltham, MA 02452 USA
tbabaian@bentley.edu

Abstract. Conformant planners solve problems with a correct but incomplete description of the initial state, by finding plans that are valid for all possible assignments of the unknown atoms. Most conformant planners, however, do not handle universal quantification, which is a problem when the set of all domain objects is unknown or very large, and thus can not be enumerated. This paper introduces PSIGRAPH, a conformant planner that operates with universally quantified statements in the initial and goal states, as well as in the action preconditions. Thus, PSIGRAPH does not need to know the complete set of domain objects. We present the algorithm and the results of its experimental evaluation, showing that PSIGRAPH is competitive with other conformant planners. PSIGRAPH is based on Graphplan, but differs from previous approaches such as Conformant Graphplan in that it does not create multiple plan graphs.

1 Introduction

Graphplan [3] is a well-known and well-studied AI Planning algorithm. From a layer of initial conditions, it iteratively generates new layers of subsequent conditions that can result from actions, detects whether these subsequent conditions entails the goal, and if so, evaluates whether the path of actions that lead from the initial conditions to the goal is a valid solution. Graphplan repeats these three steps, extending the graph until a valid solution is found. A large amount of prior work has addressed the optimization of the closed-world Graphplan, as summarized in [14].

The solution extraction portion of Graphplan has proven to be the most computationally intensive, so optimizations have included memoizing unworkable solutions, as presented in the original Graphplan paper, forward checking to detect invalid solutions in advance, dynamic variable ordering [2], and formulating solution extraction as a *constraint satisfaction problem* (CSP). Variations on the latter approach [10] attempt to construct minimized explanations of why a solution is unworkable in the form of an unworkable set of propositions at a given time step, which we refer to as a *nogood*. These nogoods are stored, and future solutions are checked via an efficient algorithm [9] to detect whether they have a nogood as a

Advances in Artificial Intelligence Theory
Research on Computing Science 16, 2005, pp. 65-76

subset. If so, the solution is not explored further. Moreover, each nogood is *regressed* [10] to previous layers, to take further advantage of it.

The above optimizations have been implemented in closed-world Graphplan based planners. In closed-world planners, all propositions are assumed to be false unless otherwise noted. There are no unknown propositions. Recent work has explored the open-world problem, where some propositions are unknown. Conformant planners do no sensing and attempt to produce a single plan that will work in every contingency no matter what is unknown. Conformant Graphplan [13] is a Graphplan-based algorithm that produces a Graphplan for each possible world. More recent planners, including GPT [4], have expressed conformant planning as a search in a belief space. MBP [7] uses Binary Decision Diagrams [6] to represent belief states. CAltAlt-LUG [5] condenses multiple planning graphs into a Labeled Uncertainty Graph to conduct the search in belief space.

However, none of the above planners handle quantified information, or information about an infinite number of items. Finzi et al [8] produced an open-world planner, implemented as a theorem prover in the situation calculus, that could represent statements like "For all Blocks x, x is not on top of A", whereas the above planners would need to make a qualitatively different statement like (Clear A). We use an open world planning language called PSIPLAN [1] that can represent quantified statements about negated propositions. Furthermore, PSIPLAN can add exceptions to these statements, such as "For all Blocks x, x is not on top of A, except if x is Block B." Babaian and Schmolze called these statements *psiforms*. Exceptions to a psiform represent unknown information. That is, given the previous statement, the state of Block B being on top of A is unknown. PSIPOP [1] is a conformant partial order planner based on PSIPLAN.

This paper describes a new planner called PSIGRAPH, which implements a Graphplan based algorithm using the PSIPLAN language, and is thus able to act as a fast conformant planner for use in domains where quantification is needed.

In the next section, we review the original Graphplan algorithm, followed by a description of PSIPLAN. Afterwards, we explain PSIGRAPH, which combines the two. We then describe the methodology used in testing PSIGRAPH on the Blocks-World and Bomb-In-Toilet-with-Clogging (BTC) domains. Finally, we evaluate the results and draw conclusions.

2 The Closed-World Graphplan Algorithm

Graphplan constructs a layered, directed, acyclic graph. The first layer is assigned level 0, and the nodes in even numbered layers represent ground literals. The nodes in odd numbered layers represent operators. No literal or operator is represented more than once in a given layer. The initial conditions are assigned to nodes in layer 0. Letting the current layer of operators be called k where initially k=1, Graphplan repeats the following steps, increasing k by 2 each time, until it finds a solution.

- All possible operators, including maintenance operators (which simply *copy* a condition from one layer to the next condition layer) are assigned to layer k.

- For each operator in level k, Graphplan checks to see whether its preconditions are present on layer k-1. If not, the operator is removed from the graph. If so, a directed edge is created from each precondition on level k to the operator.
- The effects of the operator are added to layer k+1, with directed edges from the operator to these effect nodes.
- When all operators have been examined, mutexes are created between pairs of operators that cannot co-occur. For example, a mutex would occur between two operators whose preconditions are mutually exclusive.
- Next, mutexes are created between inconsistent pairs of conditions on level k+1.
- After all mutexes have been added, Graphplan evaluates whether layer k+1 entails the goal. If so, it is possible that Graphplan has found a solution. In the next phase, called Solution Extraction, Graphplan starts with the goal conditions from layer k+1 and checks to see whether there exists a set of edges from non-mutex actions that produce them. If so, Graphplan recursively checks to see whether these actions have non-mutex conditions which produce them. If the recursion reaches the initial layer, which by definition has no mutexes, then Graphplan has found a solution.

3 PSIPLAN

PSIPLAN [1] is an expressive language designed for open world domains. It offers limited quantification and tractable, complete reasoning. A database in PSIPLAN consists of ground literals and psiforms, the latter of which express possibly quantified *negative* information. Such quantification makes a database much more compact since there are often many more false facts than true ones. For example, a briefcase may have a pencil in it but there may be many things not in the briefcase. Moreover, for infinite domains, or for finite domains where some objects are unknown to the planner, quantification is essential. Consider the impossibility of stating that there is nothing in the briefcase except a pencil if the domain is infinite, or if the domain is finite but the planner cannot name all the objects in it. In both cases, one cannot enumerate all ground instances of ~In(x,B) .

To state that briefcase B has nothing in it except possibly pencil P in it PSIPLAN uses a *psiform* [~In(x,B) except x=P]. .Here the x is a universally quantified variable and ~In(x,B) represents that no x is "in" B. The *exception* x=P means that ~In(x,B) is not necessarily true when x=P. [~In(x,B) except x=P] is equivalent to the standard first order sentence $\forall x.\neg In(x,B) \vee x = P$. Combined with the atom In(P,B), it implies that P and nothing else is in B.

Psiforms are even more general in two ways. First the main part, which is the part before the word "except", can be a clause of negated literals. For example, [~In(x,B) or ~Pencil(x)] states that "for all x, x is either not in B or x is not a pencil" -- i.e., there are no pencils in B (though there might be other things in B). Second, the exceptions can themselves be "quantified" in that a set of ground clauses can be excepted. For example, [~InDir(x,y) or ~TexFile(x)] states that "for all x and y, either x is not in directory y or x is not a Tex file," which is equivalent to saying that Tex files are not in any directory. But this is odd. A more reasonable statement might be [~InDir(x,y) or ~TexFile(x) except y=/tex], which states that Tex files are not in any directory except possibly the directory /tex.

In some domains, one can use tricks to represent quantified information without explicit quantification, such as the use of Clear(x) in the blocks world. But the use of Clear depends crucially on the requirement that there is at most one block on top of another. In the briefcase example, we cannot use a trick such as Empty(x) because a briefcase can have 0, 1, 2 or more objects in it. We would need Empty(x), Empty1(x) to represent that x is empty except for 1 object, Empty2(x), etc.

The reasoning algorithms for psiforms include entailment, logical difference and logical image. Entailment is needed because we now have quantification. For example, if our goal is that from above, namely that no block be on B, [~On(x,B) or ~Block(x)], we can satisfy this with nothing being on B, [~On(x,B)], or with nothing being a block, [~Block(x)]. Logical difference lets us "subtract" one psiform from another to see what is not entailed. For example, P1=[~On(x,B) or ~Block(x)] "minus" P2=[~On(x,B) except x=A], which states that nothing is on B except possibly A ,, yields P3=[~On(A,B) or ~Block(A)], i.e., to entail P1 using P2 we must also have P3. Image is the complement of difference. The image of P2 on P1 is the subset of P1 that is entailed by P2, which is P4=[~On(x,B) or ~Block(x) except x=A]. All three types of reasoning are used in planning.

Formally, a PSIPLAN database is a set of ground literals and/or psiforms. A psiform is $P=[\sim P_1(x)$ or … or $\sim P_m(x)$ except $\sigma_1, \ldots, \sigma_n]$ where x is possibly a vector of variables, $M(P)=[\sim P_1(x)$ or … or $\sim P_m(x)]$ is called the *main form* and $E(P)=\{\sigma_1,\ldots,\sigma_n\}$ are the exceptions. Each σ_i is a substitution that binds a (not necessarily proper) subset of the vector of variables, x, to constants. The meaning of a psiform P is the conjunction of the clauses in $\phi(P)$. When P has no exceptions, $\phi(P)$ is the set of all ground instantiations of P, i.e., $\phi(P)=\{M(P)\sigma \mid M(P)\sigma$ is a ground clause$\}$. Otherwise, $\phi(P)=\left(\phi(M(P)) \setminus \left(\phi(M(P)\sigma_1) \cup \ldots \cup \phi(M(P)\sigma_n)\right)\right)$ where \ is set difference.

A ground clause C1 entails another ground clause C2, written C1|=C2, if and only if the literals in C1 are a subset of the literals in C2. A psiform P1 entails a psiform P2, P1|=P2, if and only if every clause in ϕ(P2) is entailed by some clause in ϕ(P1). The *image* of P1 onto P2, written $P1 \rhd P2$, is the subset of ϕ(P2) that is entailed by P1. Thus $\phi(P1 \rhd P2)=\{p \mid p \in \phi(P2)$ and $\phi(P1)|=p\}$. Finally, the *e-difference* (i.e, logical difference) of P2 minus P1, written P2-P1, is the subset of ϕ(P2) that is not entailed by P1. Thus ϕ(P2-P1)$=\{p \mid p \in \phi(P2)$ and $\sim(\phi(P1)|=p)\}$. $(P1 \rhd P2)$ and (P2-P1) partitions ϕ(P2). We note that image and e-difference can be represented by a set of psiforms, and that all three operations -- entailment, image and e-difference -- require time and space that is polynomial in the size of the database under certain reasonable assumptions [1] which we make in this paper.

4 PSIGRAPH

We split the description of PSIGRAPH into four parts: the definition of a planning problem, the overall algorithm, graph generation, and solution extraction.

4.1 Definition of a Planning Problem

PSIGRAPH is given the following:

- A set of initial conditions, which consists of ground literals and/or psiforms

- A set of goals, which consists of ground literals and/or psiforms.
- A set of operators, each of which consists of:
  - a name, which specifies the variables in the operator structure.
  - a set of preconditions, which consists of literals and/or psiforms.
  - a set of effects, which consists of literals.

In the currently implemented version of PSIGRAPH, we do not allow conditional effects and disjunctions are limited to psiforms.

The overall PSIGRAPH algorithm is in Figure 1 and is the same as the closed-world Graphplan algorithm.

4.2 Graph Generation

The graph generation portion of PSIGRAPH is based on that of the closed-world Graphplan in that each precondition of each operator is checked to see if it is entailed in the previous layer. If all of the preconditions for the operator are so entailed, the operator is retained and the effects of the operator are generated for the next layer. Otherwise the operator is removed from the graph. However, there are three issues presented by the use of psiforms in the PSIGRAPH domain.

(1) Preconditions may be *nearly entailed* by propositions.
(2) There may be more than one way to entail a precondition.
(3) Generated psiforms on the next layer may only be *partially mutex* with other generated psiforms, and this will make future reasoning difficult.

We explain each of these in turn. But first, we say that a psiform P1 *nearly entails* another psiform P2 if and only if the main part of P1 entails the main part of P2, ignoring exceptions, i.e., P1 nearly entails P2 iff M(P1)|=M(P2).

```
Algorithm PSIGRAPH
    Current-Level = Initial-Conditions; Iterations=0
    Repeat
        Iterations++;
        Next-Level = Generate-New-Layer(Current-Level);
        If Find-Plan(Next-Level) == SUCCESS
            then Return(SUCCESS);
        End if
        Next-Level = Current-Level;
        If iterations > MAX_ITERATIONS, Return(FAIL);
    end Repeat
```

Fig. 1. The overall PSIGRAPH algorithm.

4.3 Multilinks

The first issue arises when a combination of two or more propositions from a layer entail a precondition or goal, but neither by itself is sufficient for such entailment. For example, let a precondition state that block B is clear of anything on top, i.e., [~On(x,B)], and let the previous layer include the propositions:

[~On(x,B) except x=C, x=D], ~On(C,B), ~On(D,B)

Together, these three conditions entail the precondition. In such a case, PSIGRAPH draws a *multilink* between the operator and the three prop ositions. A multilink in PSIGRAPH acts just like a link or an edge in Graphplan. It is a set of edges from one or more propositions on layer k to an operator on layer k+1. As a plan proceeds these edges must be followed atomically, that is, all at once or not at all. Note that the closed-world Graphplan may be viewed as a form of PSIGRAPH where all the multilinks have exactly one edge.

4.4 Finding the complete set of Multilinks

The second issue is that a precondition may be entailed by more than one multilink. For PSIGRAPH to be complete, it must find all possible multilinks. Thus it implements the function Satisfy-Goal, which returns the set of sets of propositions in a given layer where each set, taken together, entails a given goal. It *e-subtracts* each potentially helpful proposition from the goal, and recursively calls itself to satisfy the remainder. The algorithm is in Figure 2 where \ is set subtraction and – is e-difference.

The first argument to the recursive call is the union of the set of goals without G and the e-difference of G minus P. The latter is the portion of G that is not entailed by P. In general, e-difference returns a set of psiforms.

```
Function Satisfy-Goal (Goals, Props, Sofar)
- Goals is a set of psiforms to achieve.
- Props is the set of conditions to examine.
- Sofar is the current partial solution set.
If Goals is empty then return {Sofar}
        // Return a set whose only element is the set Sofar.
else Let Result = {}
                For each P in Props
                        For each G in Goals
                                If P nearly entails G
                                then Result=Result U
                                        Satisfy-Goal((Goals\G) U (G-P),
                                                Props\P, SoFar U {P})
                        end inner for
                end outer for
        return Result
end If
end Function
```

Fig. 2. Satisfy-Goal

The first call for a given Goal is: Satisfy-Goal({Goal}, Props(Layer), {})
where {Goal} is a singleton set containing Goal, Props(Layer) is the set of propositions in the Layer and {} is the empty set.

4.5 Partially Mutex

Just like Graphplan, PSIGRAPH generates all of an operator's effects on the next layer. For negated literals, determining mutexes between conditions on this next layer is the same as Graphplan since all literals are ground: If A is an atom, mark as mutex the pairs A and ~A. With psiforms, mutexes are more complicated because an atom A might be inconsistent with only *part* of a psiform P, i.e., P might entail many ground clauses where A is inconsistent with only some of them. For example, A=On(A,B) is inconsistent with P=[~On(x,B)] but P entails many ground literals besides the one that is inconsistent with A. We cannot mark A and P as mutex because it is an overgeneralization and may prevent finding some solutions. Instead we *split* P into two parts: P1, which represents the subset of P that directly conflicts with A, and P2, which is the remainder of P. In the above example, we split P into P1=[~On(A,B)] and P2=[~On(x,B) except x=A].

The above is accomplished using the image and e-difference operation described earlier. An atom A is inconsistent with a psiform P iff P⊨(~A). If not, there is no mutex. If so, we calculate P1=([~A] ⊳ P) and P2=(P1-[~A]). Remember that P1 and P2 are sets of psiforms, and we note that P1 must be a singleton set. If P2 is empty then no splitting occurs because [~A] entails all of P. In this case, A and P are simply marked mutex. If P2 is not empty, then node P is replaced by P'=(P1 U P2) in the graph and A is marked mutex with the single psiforms in P1. P1 and P2 inherit the uplinks from P. Their downlinks are easily recalculated from P's downlinks.

4.6 Solution Extraction

Solution extraction of PSIGRAPH follows the algorithm of Kambhampati [10] by using Explanation-Based Learning (EBL) and Directed-Backtracking (DDB). Several issues that arise due to the use of psiforms in PSIGRAPH require only minor modifications to the algorithm

(1) There may be more than one set of propositions that entails a goal or precondition.
(2) A set of propositions may be mutex, even though there is no pairwise mutex. We refer to these sets as nogoods.

The first issue is solved merely by following all possible multilinks backwards during backtracking. Although this increases the search space, the EBL/DDB algorithm is extended to mark additional sets of unreachable propositions as memoizations of nogoods. The only difference is that in PSIGRAPH, a failed solution could return more than one conflict set. Each conflict set is stored as a memo and regressed. The memo sets are stored in a UB-Tree [9].

The second item above refers to disjunctive psiforms. A disjunctive psiform may be mutex with a pair of atoms taken together, while being mutex with neither separately. These sets are detected at graph generation time by scanning the layer for sets of atoms each of which is mutex to a term in the disjunction. They are stored as nogoods in the UB-tree.

5 Evaluation

PSIGRAPH was implemented in Allegro Common Lisp, and tested on the BTC (bomb in toilet with clogging [11]) and Blocks-World domains. For the Blocks-World domain, we generated problems using the BWStates program [12] and recoded them in PSIGRAPH. For BTC, we rephrased the initial conditions as follows. In this example, there is one toilet, T1, and 2 packages, P1 and P2.

[~Package(x) except x=P1, x=P2],
Package(P1), Package(P2), Toilet(T1), ~Clogged(T1)

The first proposition states that nothing is a package except possibly P1 and P2. We also rephrased the goal.

[~Package(x) or ~Armed(x)]

i.e., every x is either not a package or not armed. The Dunk(P,T) action had preconditions Package(P) and ~Clogged(T), and effects Clogged(T) and ~Armed(P). The Flush(T) action had no preconditions, and effect ~Clogged(T).

We also performed experiments where it was not known whether the toilet(s) were clogged (i.e, we removed ~Clogged(T1), etc., from the initial state), and the effect was small. We will soon see that PSIGRAPH is not sensitive to this type of change in the initial state. We ran our experiments on a 2.4Ghz Dell Linux workstation.

We used two different versions of PSIGRAPH. The first, PG1, performed an exhaustive solution extraction search on each layer before failing and proceeding to the next layer. PG1 always finds an optimum parallel solution. The second, PG2, differs from PG1 in the following ways:

- (Mod 1) All pairs of non-maintenance actions were labeled mutex.
- (Mod 2) Solution extraction failed after n nogoods were found, where n is the number of operators in the domain, unless the number of planning layers was at a theoretical maximum (in which case solution extraction failed). The last solution extraction performed before PSIGRAPH gives up is always a complete search.
- (Mod 3) Solution extraction was only attempted every fifth layer.

(Mod 1) means that PG2 finds only linear plans. Problem BTC(40,6) requires 13 time-steps under PG1 and 81 timesteps under PG2, although both have the same number of non-maintenance operators.

(Mod 2) prevents the planner from getting bogged down in solution extractions that are likely to fail. As a result, it may return non-optimal plans. But as long as the last attempt is a full solution extraction, it will never fail to solve a plan because of (Mod 2). This is because solution extraction works just fine on overly long graphs. BTC was not assigned a theoretical maximum, but the Blocks-World domain has a maximum number of plan steps of 2 times the number of blocks

(Mod 3) has the same intention as (Mod 2).

Table 1 shows our results in BTC 1-toilet problems. BTC 1-toilet results have been published for other conformant planners, and a summary in [5] includes results for CAltAlt-Lug [5], HSCP, GPT [4], and CGP [13]. The summary shows HSCP as the fastest timing on this domain, taking 98 seconds for the 20 package problem, 674 seconds for 40 packages, and 5100 seconds for 60 packages. We note that these planners allow conditional effects but not quantified information, whereas PSIGRAPH does not allow conditional effects, but does allow limited quantified information. The effect is that the difference in expressiveness helps make BTC an easier problem for PSIGRAPH, as the DUNK action has no preconditions that need to be explored.

Table 2 compares the BTC 10-package 3-toilet problem (BTC(10,3)) and BTC(40,6), where the possible clogging of all toilets was unknown, to published results of WSPDF [8], who used a 333 MHz Sun Sparc 10 Ultra workstation. Finzi et al use a domain dependant BadSituations marker to limit their search space. In WSPDF, a BadSituation occurs when a toilet is flushed twice without an intervening dunk, when a package is dunked when there is an undunked package lower in number, and when a toilet is flushed when there is an unflushed toilet lower in number. We did not use the

Table 1. Timings of various planners on various domains. Times are in seconds. PSIGRAPH was run 5 times on a 2.4 Ghz Pentium processor. All other results come from [5] on a 2.66 Ghz Pentium 4. Times are in format (x/y), x is in seconds, y is in plan steps. All plans in the same row produce the same number of plan steps, unless otherwise noted. * indicates no solution

Domain	PG1	PG2	Caltalt Lug	HSBP	CGP
BTC (20,1)	2.46/39	1.7/39	651	98	465/3
(40,1)	*	14.4/79	8009	674	*
(60,1)	*	80.2/119	38393	5100	*

Table 2. Timings in the BTC domain for multiple toilets with high uncertainty. PG1 and PG2 were run using a 2.4 Ghz Pentium 4 processor. WSPDF is reported from [8] on a 333 Mhz UltraSparc 10. * indicates no solution.

Domain	PG1	PG2	WSPDF	
BTC(10,3)	12.5/7	1.2/19	.32/20	
BTC(40,6)	*	80.3/79	114/80	

Table 3 . Averaged results of running PSIGRAPH on 10 random examples in the BW domain. Domains are of the form BW(a,b) where a is the number of blocks and b is the number of blocks whose location is unknown. Results are of the form x(y)/z, w here x is mean time in seconds, y is mean plan steps,. And z is the maximum number of propositions found in a single layer. '*' indicates a trial had no solution found after 10 minutes, '-' indicates the experiment was not run

Domain	PG1	PG2
BW(8,0)	3.9(3.6) / 136	21.6(4.3)
BW(10,0)	28.2(5) / 210	*
BW(11,0)	81.7(5.7) /253	*
BW(12,0)	334.1(4.1)/300	*
BW(15,5)	37.8(4.4) /210	-
BW(20,10)	54.1(5.2)/210	-

.

above domain restrictions but we did try to limit the search space in PG2 (see above).

As the results show, PG1 produces optimal solutions, even on multiple-toilet problems. This is because it performs a complete search of the solution space. Its

disadvantage is that it spends large amounts of time performing failed solution extractions, and this is enough to make the planner time out for large problems.

PG2, by contrast, finds solutions much faster. The speed of PG2 is in part an artifact of the simplicity of the BTC domain, as PG2 does not need to spend much time at all in solution extraction. In these experiments, PG2 was dominated by the graph generation phase, a trend that would reverse itself on more difficult problems. Graph generation is a comparatively easy task whereas solution extraction requires searching an exponential number of possible solutions. Furthermore, if the metric of finding n mutexes per attempt at solution extraction (Mod 2, where n is the number of literals on the layer) makes little progress on each iteration, PG2 might take longer than PG1. Also, PG2 relies on the hope that it will find a solution without exploring the whole search space. We ran PG2, for instance, reversing the order that the operators are considered (that is, we tried preferring maintenance actions instead of preferring non-maintenance actions), and PG2 showed the same difficulties for larger BTC domains as PG1. Thus, PG2 may prove to be fragile on other domains. The results above for PG2 should be viewed as an optimistic scenario, not the expected scenario. PSIGRAPH is presented with a similar dilemma to that faced by a closed-world Graphplan with a large number of propositions. We note that the BTC domain will generate approximately 2*P propositions per layer, where P is the number of packages, as there are P initial conditions of the form (Package P), and P exceptions to [~Package(x)].

We ran PG1 and PG2 on the Blocks-World domain to test the algorithm in a more difficult domain as well as to test its sensitivity to the number of unknowns in the initial state. We used the BWStates program [12] with various numbers of blocks with various numbers of unknown locations. Each problem was translated to PSIPLAN, including elimination of Clear and use of psiforms instead. Table 3 shows the tradeoff between PG1 and PG2.

Table 3 shows that PG1 is better on domains like blocks-world, presumably because in the blocks-world doing an exhaustive solution extraction early and often is a good idea since more mutexes will be found anyway. It also shows that PSIGRAPH is relatively insensitive to unknowns in the domain. Domains (15,5) and (20,10) are comparable to (10,0), in that both have the same number of known facts in the initial state. Unknowns will not affect solution extraction; they only affect the time taken for graph generation as they increase the number of operators to check. It should be noted that Finzi et al. also timed their WSPDF theorem prover on the Blocks-World domain, with a domain-specific BadSituation() predicate which favored exploration of good towers. Their planner produced 17-step plans in 31.2 seconds, with an additional 50.1 seconds to compile the domain for 20 blocks and 10 unknowns on a 333 Mhz UltraSparc processor. These results are roughly comparable to ours. However, PSIGRAPH did not rely on any additional domain information, like the BadSituations..

8 Conclusion

We introduced PSIGRAPH, a conformant planner based on Graphplan that uses the PSIPLAN language, which allows for limited quantification. PSIGRAPH can work in

infinite domains, and in finite domains where not all objects are known and admits very compact representations of domains with a large quantity of negative facts.

We evaluated PSIGRAPH on the BTC and Blocks-World (BW) domains, and compared results from other planners. Only one of these other planners allows quantification, namely WSPDF of [8]. For several BTC problems, PSIGRAPH is faster than most other planners tested. In BW, PSIGRAPH is comparable to WSPDF, though it is not clear how WSPDF's domain dependent BadSituations affected its timings.

Future work will improve PSIGRAPH, investigate more domains, and develop a better understanding of the differences between PG1 and PG2. We will also expand PSIGRAPH to allow conditional effects and general disjunction in the initial state, and will explore the use of binary decision diagrams [6] for both ground and quantified formulas.

References

1. Babaian, T. and J. Schmolze (2000). PSIPLAN: open world planning with psi-forms. In Artificial Intelligence Planning and Scheduling: Proceedings of the Fifth International Conference (AIPS'00), pages 292-300.
2. F. Bacchus and P. van Run (1995). Dynamic variable ordering in csps. In Proceedings of the 1995 conference on Principles and Practice of Constraint Programming pages 258-275, September 1995.
3. Blum and M. Furst (1995). Fast Planning through planning graph analysis. In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95), pages 1636-1642, 1995.
4. Bonet and H. Geffner. Planning with Incomplete Information as Heuristic Search in Belief Space. In AIPS-2000, pages 52-61, 2000.
5. Bryce, D., Kambhampati, S (2004)., and Smith, D.E. AAAI 2004 workshop on Learning and Planning in Markov Decision Processes, 2004.
6. Bryant, R.E. (1986). Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, C-35(8), 677-691.
7. Cimatti, A. and Roveri, M. (1999). Conformant planning via model checking. In Biundo, S. ed., Proc. ECP99: European Conference on Planning. Springer-Verlag.
8. Finzi, Pirri, and Reiter (2000). Open World Planning in the Situation Calculus. In Technical Report, University of Toronto. AAAI, pages 754-760, 2000.
9. Hoffman and Koehler (1999). A new Method to Index and Query Sets. In 16th IJCAI, pages 462-467, 1999.
10. Kambhampati (2000). Planning Graph as a (Dynamic) CSP: Exploiting EBL, DDB and other CSP Search Techniques in Graphplan. Journal of Artificial Intelligence Research 12 (2000), pages 1-34.
11. McDermott, D. A critique of pure reason. Computational Intelligence, 3(3):151-237, 1987.
12. Slaney, J., and Thiebaux, S. 1996. Linear time near-optimal planning in the blocks-world. In Proc. Thirteenth National Conf. on Artificial Intelligence, 1208-1214.
13. David E. Smith and Daniel S. Weld. Conformant Graphplan. In (AAAI-98) and IAAI-98), pages 889-896, Menlo Park, July 26-30, 1998. AAAI Press.
14. Weld, D. S. (1999). Recent advances in AI planning. AI Magazine 20(2), 93-123.