Comparison of Methods for Improving Search Efficiency in a Partial-Order Planner *

Raghavan Srinivasan **Adele E. Howe**

Computer Science Department
Colorado State University
Fort Collins, CO 80523
Net: {srinivas,howe}@cs.colostate.edu

Abstract

The search space in partial-order planning grows quickly with the number of subgoals and initial conditions, as well as less countable factors such as operator ordering and subgoal interactions. For partial-order planners to solve more than simple problems, the expansion of the search space will need to be controlled. This paper presents four new approaches to controlling search space expansion by exploiting commonalities in emerging plans. These approaches are described in terms of their algorithms, their effect on the completeness and correctness of the underlying planner and their expected performance. The four new and two existing approaches are compared on several metrics of search space and planning overhead.

1 Improving Search Efficiency in Planners

Partial order planning is becoming a common method of planning. Unfortunately, but hardly unexpectedly, the search space in partial order planning expands quickly as the problem size increases. Unfortunately, but less expectedly, search space expansion is dependent on a variety of factors, some of which are difficult to predict. A problem that was solved in short order may be made impossible to solve in reasonable time simply by adding an innocuous looking new goal, by changing the ordering of goals or even by adding a few more objects to the problem initial state.

More graceful degradation of performance can be achieved by identifying aspects of the planner most susceptible to the problem changes and developing methods to ameliorate the search space expansion. This paper presents four new approaches for improving efficiency in a partial order planner by exploiting commonalities between proposed plans during two phases of planning: flaw selection and plan refinement. These approaches are described in terms of their algorithms, their effect on the completeness and correctness of the underlying planner and their expected performance when compared to two existing approaches. Finally, the new and existing approaches are compared on several metrics of search space and planning overhead.

The goal of this project was to determine why it could be so hard to design efficient problem descriptions for UCPOP, a partial order planner. We found that what seemed like trivial problems could not be solved in reasonable time. Indeed, minor variations on the same problem led to UCPOP's being unable to solve the new problem. Using a variety of analysis methods, we determined that the primary fault lay in UCPOP's selection of flaws to repair and additions to the plan to repair the flaws. Three approaches have been used to enhance search in planning: more sophisticated plan representations and reasoning (e.g., hierarchical planning and resource reasoning as in [Tate *et al.*, 1994; Wilkins, 1988]), domain specific control knowledge (through programmer intervention or machine learning) and improved search methods. We focused on the third approach because it requires the least change to the underlying planner.

We used UCPOP because it is an easily available, domain independent partial order planner [Penberthy and Weld, 1992]. UCPOP plans by iteratively selecting and repairing flaws in the current plan. A flaw is repaired by adding steps and constraints to the plan. The search control strategy decides which partial plan to select for expansion. In general, UCPOP gives good results on small domains and problems in which subgoals are independent. For problems with interrelated subgoals or those requiring arithmetic, UCPOP often does not find a solution even with very large search limits.

The Least Cost Flaw Repair (LCFR) strategy [Joslin and Pollack, 1994] improved search control in UCPOP by selecting the flaw with the minimum repair cost. The repair cost of a flaw is defined as the number of plans

*This research was supported by a National Science Foundation Research Initiation Award #RIA IRI-9308573 and ARPA-AFOSR contract F30602-93-C-0100. We also wish to thank the anonymous reviewers for their suggestions for improving content and clarity.

generated to repair it. Open conditions and threats are treated alike. The main drawback of LCFR is the overhead incurred for flaw selection. The total time spent in planning with LCFR can be more than that for UCPOP, even though UCPOP examines far more plans than LCFR. However, LCFR reduces the search space more than other flaw selection strategies [Peot and Smith, 1993].

A variant on LCFR, QLCFR [Joslin and Pollack, 1994], assumes the cost of un-repaired flaws to be constant over time; it caches the results of estimating flaw repair costs and uses the cached cost as the estimate in subsequent flaw selection. QLCFR reduced the overhead of LCFR, but at a cost of solving fewer problems.

2 Similar Flaws and LCFR (Templates)

LCFR is expensive because it estimates separately the cost of repair for every flaw in every potential extension to the current plan. However, in most problems, flaws can be similar; they involve the same type of condition and are amenable to repair by the same fix. For example, flaws in the Blocks World domain are commonly of the form `(on ?x ?y)` or `(clear ?x)`. The resolution of any flaw of these forms is likely to be the same (e.g., add an action to move the indicated block); hence, we can expect the cost of flaw repair to be *roughly* the same for flaws with similar forms.

Consequently, we exploit the similarity in flaws to reduce the number of repair cost estimates to be made. In particular, we assume that at a particular stage of plan refinement, the repair cost is the same for all similar flaws. Other than this approximation, repair cost is the same as that of LCFR scheme.

QLCFR also approximated the repair cost of flaws by estimating once and re-using the estimate. The difference between our approach and QLCFR is that QLCFR cached the estimate and re-used it in *subsequent* plan refinements rather than applying it to similar flaws at the same point in plan refinement. Our approach allows recently acquired information to be incorporated in estimating cost.

The first step towards deciding how to change a developing plan is to identify and group together identical open conditions in the plan. Two open conditions are said to be *similar* if they have the same predicate. For example, `(pred1 ?x)` and `(pred1 ?y)` are similar. A set of similar open conditions with predicate `p` are said to form a template `p`. All of the open conditions in a plan can be grouped into a set of templates.

We assume that when open conditions are similar, the order in which they are selected for repair does not matter. Thus, the repair cost of a template is estimated by finding the repair cost of the first member of the template.

This approach approximates only the cost of open conditions. Threats are not easily grouped because they do not involve variable bindings. Thus, similar threats often do not have similar resolutions and are often resolved as a side effect of repairing some other flaws. Consequently, a uniform repair cost would not be a reasonable approximation of the actual costs.

Open conditions are considered only if a plan does not have any threats. If a plan has threats, the one with the minimum repair cost is selected; otherwise, the first member of the template with the minimum repair cost is selected.

2.1 Expected Performance

We expected that the average number of plans examined before finding a solution in this scheme should be comparable to that of LCFR, while the overhead should be much less than that of LCFR. Overhead is defined as the number of extra plans created in service of estimating flaw cost. Since only a subset of the open conditions are evaluated in the templating approach, on the average, its overhead should be less than that of LCFR. However, in the worst case, the templating approach can incur more overhead than LCFR when the estimate for one member of the template does not generalize to the rest; potentially causing additional backtracking. Empirical performance is reported in Section 6.

2.2 Correctness and Completeness

Because only flaw selection is modified, the correctness and completeness of UCPOP is maintained by the templating approach.

3 Templates with Repair Reuse

With Templates, open conditions are grouped to estimate repair cost. We extend this idea to the next step: selecting (or *reusing*) similar actions to add to repair similar flaws. Consequently, given that an action is added to the plan to repair a flaw of a particular type, another instance of the same action can be added, at the same time, to repair another flaw of the same type. This sense of reuse is much more limited and local than what is typically meant by plan reuse (e.g., [Kambhampati and Hendler, 1992]); it is constrained to reusing the occasional step within a plan being developed.

Consider a plan P with a set of flaws F. F can be grouped into a set of templates $T = \{T_1, T_2, \ldots T_m\}$. Each T_i consists of a set of similar flaws. Let T_{min} ($1 \leq min \leq m$) be the template with the minimum repair cost. The first flaw in the T_{min} set, $f_{min,1}$, is selected for repair, and a set of new plans P' are generated. Let P'_s be a subset of P' such that each plan in P'_s includes a new plan step for repairing $f_{min,1}$. For each plan in P'_s, a set of new plans are generated in which some of the flaws of type T_{min} are repaired by adding another instance of an action added for $f_{min,1}$.

Two values are returned: the plans in which all flaws of type T_{min} are repaired by adding the same type of action, and the plans in which some but not all flaws are repaired this way. We require the second value to facilitate backtracking. As with the basic templating scheme, not all flaws of a similar type require the same cost or action for repair (e.g., some might be satisfied by initial conditions). Consequently, the first set of plans are added to the search queue and the second set is stored in the event of later backtracking.

3.1 Expected Performance

Two opposing factors were expected to affect the performance as measured by plans examined and overhead. If reuse is successful most of the time, then both plans examined and overhead will be less; however, if new threats are introduced due to reuse, repairing them will cost more in terms of plans examined as well as overhead. The worst case will occur when an early attempt to reuse is inappropriate, leading to considerable backtracking. As a consequence, we expected the success of this approach to be highly problem/domain dependent.

3.2 Correctness and Completeness

Any newly added plan step may introduce threats; for each of the flaws in template T_{min}, a set of new threats could be introduced. However, all of these introduced threats will be detected. Thus, the final solution will still be correct. In addition, the backtracking facility insures that if a solution exists, it will eventually be found. Consequently, completeness and correctness are maintained.

4 Probabilistic Reuse

Templating and Reuse can be viewed as approaches in which plan repair reuse is applied with probability 0 and 1 respectively. Because we suspect that plan repair reuse is not always the best strategy (and cannot currently recognize when it is and is not the best strategy), we can define an approach in which reuse is applied with some probability p, $0 < p < 1$. Intuitively, some p exists for which the performance will be better than that of Templating or Reuse. This value can be determined empirically. Obviously, the value of p depends both on the problem and domain. We hypothesize that p should be small non-zero value, and so determined it empirically. For all tests, the same value of p, 0.2, was used.

5 Adding a New Construct to the Plan Language (Bang-UCPOP)

The previous approaches all altered the control of plan expansion within the planner only. One alternative is to make the plan language more expressive of constraints known by the user. A simple constraint is that multiple inclusions of the same operator within a single plan should be instantiated to *different* objects within the environment. This hard constraint is a simple form of the resource reasoning included in more sophisticated planning systems.

We developed this approach to address problems discovered when analyzing the behavior of UCPOP in Truckworld [Hanks *et al.*, 1993] (a simulator of trucks moving cargo between different destinations). UCPOP fails (i.e., could not find a plan even given a large search space) on apparently simple conjunctive subgoal problems in Truckworld. A typical example is "Bring 4 fuel drums from outside the truck and fill the fuel tank." Because the size of the search space increased dramatically with the order and number of identical subgoals, we hypothesized that the number of identical fuel drums needed and available might lead the planner to search unnecessarily for the *right* binding of fuel drums in the *right* order.

We studied the behavior of UCPOP in Truckworld by collecting execution traces of UCPOP working on Truckworld problems with similar conjunctive sub-goals. Using CLIP [Anderson *et al.*, 1993] (an instrumentation tool for defining and running data collection routines in a simulated environment), we collected data on what plans were generated, how certain open conditions were repaired, what threats were considered, and what variable bindings were used.

We analyzed the data with a variety of methods, from simple eyeballing through dependency detection [Howe and Cohen, 1994], and determined that, in effect, UCPOP was searching in circles: trying the same variable bindings over and over again. For example, consider the problem of picking up two identical fuel drums from a world which has five such drums. To repair the first open condition (i.e., picking up the first drum), a set of five possible plans are generated. For the second drum, a similar set of plans is generated, with one of them trying to reuse the first step to get the first drum. This results in a threat. Next, UCPOP tries binding a new value for the first flaw. It continues to try pairs of identical bindings before it finds two unique binding values that can repair both the open conditions. Most of the search time is wasted in trying the same values for variables that require different values. Thus, the plan language needs a construct to indicate to UCPOP that it should use different variable bindings for certain variables, so that it can converge on the solution much faster.

5.1 Scheme Description

For this scheme, a new language construct which creates a "special variable" is introduced. Bindings of such a variable are treated differently; in particular, the planner will *ensure* that if a binding value is needed for the special variable it will differ from that used in all previous instances of this operator in the current plan. Moreover, if more than one of such bindings are possible, only one plan using exactly one value is created; plans for other possible unique values are saved in the event of backtracking.

A special variable is denoted by the prefix '?!' (hence,

```
(define (operator pick-drum)
   :parameters   (?amt ?!pos ?arm)
   :precondition (and (outside ?arm)
                      (drum-at ?!pos ?amt))
   :effect       (and (not (drum-at ?!pos ?amt))
      (amount-in-arm ?arm ?amt)))
```

Figure 1: UCPOP operator for Truck World that illustrates the use of a Bang variable, `?!pos`

the name *Bang-UCPOP* for this approach). Bang variables are treated differently only during binding. Currently, only one such variable per plan operator is allowed, in order to minimize the complexity of resolving which variable binding resulted in a threat. Another restriction is that two operators that clobber each other should not use the same type (as defined by the plan domain) of special variable.

Special variables have a curious but useful side effect on repairing threats. For example, given two instances $O_{si,1}$ and $O_{si,2}$ of the same operator O_{si} and let p_s be the special variable parameter in its operator, then the new scheme ensures that unique values will be used for p_s in $O_{si,1}$ and $O_{si,2}$. Under the normal planning process, an unsafe link may be introduced due to $O_{si,2}$, but now there is no threat. Hence, the planner marks this threat as bogus and removes it. This saves time that otherwise would be wasted on resolving such threats.

Figure 1 shows an example of an operator which uses a bang variable. The operator comes from the Truckworld domain and is one of the operators needed to refuel a truck. The bang variable, `?!pos`, indicates the position at which the fuel drum is stored. Multiple fuel drums are typically required to refuel a truck; thus, a plan may include multiple instances of this operator, each referring to a different fuel drum in a different location. Bang is ideal for this situation because we do not wish to attempt to pick up the same fuel drum repeatedly during refueling; we can only gainfully empty it once.

Unlike the other approaches, this approach required considerable change to the algorithm for linking in new actions to plans. To expedite backtracking, the algorithm caches alternative unique variable bindings and search control maintains two search queues. When a planning failure occurs, it moves a plan from the most recent backup list into the primary search queue and continues. The modified algorithm is shown in Figure 2.

Plan language constructs for restricting search space are available in some hierarchical planners. For example, O-Plan2 [Tate *et al.*, 1994] uses condition types, which allow the domain writer to restrict selection of actions as well as to bind variables. The 'only_use_for_query' condition type of O-Plan2 resembles the Bang scheme, but differs in the situations for which it is the best approach. The Bang scheme is most effective when the

```
PLAN-LINKING(open-cond, step, current)
plan-list := NULL
more-plans := NULL
; let V be the variable in open-cond to be bound.
While binding-exists(V)
    if (special-variable(V))
        ; find a binding not used in other instances
        B := unique-binding(V)
        ; if a binding can be found, generate plans
        if (B != NULL)
            current := make-plan(B,open-cond,current)
        else current := NULL
        ; add to plans for backtracking
        if (plan-list !=NULL)
            more-plans := add(current,more-plans)
            current := NULL
    else ; find a binding with normal methods
        B := binding(V)
        current := make-plan(B,open-cond,current)
    if (current != NULL)
        plan-list := add(current,plan-list)
; return current plan and list for backtracking
return plan-list, more-plans
```

Figure 2: Algorithm for linking in new plan actions under the Bang-UCPOP approach

number of binding values is large and no one is preferred. Only_use_for_query cannot be applied in specific actions and does not look for previous bindings used in other instances of the current action. An over-indulging O-Plan2 condition type can result in the planner throwing away valid plans, whereas Bang stores all plans for later backtracking. The Bang scheme can be modified to selectively recognize bang variables at the problem level. In O-Plan2, the condition type information is built into the domain specification.

5.2 Expected Performance

Best case performance, in terms of number of plans examined, occurs when problems have identical conjunctive subgoals and when the first variable bindings do not need to be retracted later. The worst case performance occurs when the unique values selected early do not satisfy all the subgoals, thus requiring backtracking. This approach is expected to do much better than other approaches for domains with many possible bindings to the same variables, as in the motivating Truckworld example. In other cases, this approach may incur additional backtracking and thus additional computation because the new constraint does not help.

The major drawback of this approach is that it requires user intervention. The user must know when to use bang variables in a domain description (e.g., when it is expected that problems will contain multiple conjunctive sub-goals involving the same types of objects).

5.3 Correctness and Completeness of Approach

To make sure that we have not violated the correctness and completeness of the underlying planner, we need to prove that when special variable operators are used, every answer is a correct solution to the planning problem and that if a solution exists it will eventually be found.

The proof consists of three parts:

1. Even though the algorithm is limited to only one binding value for a special variable, backtracking is still permitted and thus completeness is preserved.
2. When special variables are bound to values from goal terms, then correctness is preserved.
3. When special variables are bound to particular unique values, marking threats as bogus when they are due to different instances of the same special variable operator does not affect correctness.

The correctness and completeness of UCPOP has already been proven [Penberthy and Weld, 1992], so we will show that all these cases are reducible to UCPOP. If UCPOP cannot find a solution (e.g., if enough unique values do not exist), then neither can our modification. A complete proof is beyond the scope of this paper (see [Srinivasan and Howe, 1995] for details), but we can provide a sketch of each part.

Part 1: Backtracking If a special variable is included in a new plan refinement, then the inclusion will cause a single new plan to be added to the search queue with all other possible plans being put onto a "reserve" queue. Should later plan refinements led to a failure, then the next possible plan from the reserve queue can be moved into the search queue and plan refinement continued from there. Thus, no potential plans have been pruned irretrievably; backtracking and thus completeness is preserved.

Part 2: Goal Terms When special variables are bound to values from goal terms, then no searching needs to be done for variable bindings. Thus, the operator incorporating the special variable is treated just like other operators, and correctness, as in the original scheme, is preserved.

Part 3: Bogus Threats In UCPOP, threats are detected when two conditions in the current plan have the same predicate (e.g., the "clear" condition from Blocks world). In Bang-UCPOP, if the threat involves a special variable that was not bound as part of the goal term (whose correctness was proven in part 2), we know that no such threat actually exists because the two conditions have been instantiated to different variables. Therefore, such a threat can be marked as "bogus". This does not affect backtracking because if the variables are not special then the normal rules of binding in UCPOP hold.

6 Comparison of Approaches

In this paper, we have defined four extensions to two current approaches (vanilla UCPOP and LCFR in UCPOP) for controlling plan search in a partial order planner. We expected the new approaches to perform significantly better than LCFR or UCPOP in some domains/problems. The goal of the comparison was to determine which of the six approaches works best in some common planning problems.

Three performance metrics were collected: number of plans examined before reaching a solution, overhead incurred in terms of the number of plans created for flaw selection, and CPU time. On average, we expected that the four new approaches, templating, reuse, probabilistic reuse and Bang-UCPOP, would compare favorably to LCFR on plans examined but would have less overhead and so require less CPU time. A complete report of results in provided in [Srinivasan and Howe, 1995].

6.1 Experiment Design

The six approaches were tested on 40 problems in ten domains. The same set of problems without any modification is used for all versions. Most of the problems are from the example domains provided with UCPOP and tested in Joslin and Pollack's research with LCFR. Four of the problems are from the Truckworld domain [Hanks *et al.*, 1993], all of which require picking up fuel drums; the four differ in the number of subgoals and arm positions. In all the domains, some of the operators were modified to include a special variable parameter for Bang-UCPOP. Because most of the domains are small in size, only one special variable operator was used. All trials were run on the same SPARC IPX workstation in the same version of Common Lisp.

For all cases, the search limit was restricted to 10000 plans examined. A failure was reported only when no possible plan could be found within that limit.

6.2 Results

The results are reported in Tables 1 thru 4. Table 1 presents the number of problems within each domain that were solved by each approach. The domains were: Blocks World (A), Truck World (B), Robot Domain (C), Monkey and Banana (D), Briefcase World (E), Russell's Tire World (F), Fridge Domain (G), Strips World (H), Office Domain (I), and Others (J). Table 2 lists the average number of plans examined by each approach in problems within each test domain; this corresponds to how much of the space was explored during plan refinement. Table 3 lists the average number of plans created for flaw selection (which included those created to estimate cost) for each approach in each problem domain; UCPOP and Bang are not included because they do not

	A	B	C	D	E	F	G	H	I	J	
UCPOP	4	3	2	2	4	4	1	0	7	3	30
LCFR	4	4	2	2	4	5	2	1	7	6	37
Temp.	4	4	2	2	4	5	1	0	7	6	35
Reuse	3	4	2	2	4	6	0	1	7	6	35
Prob	4	4	2	2	4	6	0	1	7	6	36
Bang	4	4	2	2	4	5	1	0	7	6	35
All	4	4	2	3	4	6	2	2	7	6	40

Table 1: Number of problems solved by the search control strategies.

	LCFR	Template	Reuse	Prob Reuse
A	1669	6566	6157	5265
B	28772	11802	1872	2529
C	8076	3535	1034	322
D	105518	5558	5558	5558
E	1645	847	563	647
F	59008	11036	3760	2547
G	9754	15075	42466	42466
H	258897	79230	48030	79230
I	459	257	267	267
J	78080	11697	11789	11625

Table 3: Average Overhead, number of plans created for all problems.

create any plans for flaw selection. Finally, as a crude estimate of both factors incorporated in the previous two measures and those not, average CPU time is provided in Table 4.

Table 1 shows that LCFR solves the largest number of problems. However, the four new approaches solve all but one or two of those solved by LCFR. All approaches solve considerably more problems than UCPOP.

In terms of number of plans examined, we expected the performance of the four new approaches to be comparable on average to LCFR and better than UCPOP. The data (in Table 2) shows that the average case performance is comparable in about half the domains, with the "best" average (numbers in boldface) for each domain distributed among the approaches. In all but a few cases, LCFR and the four new cases offer either a comparable number of plans examined or a reduction over UCPOP.

While plans examined was expected to be comparable or worse than LCFR, we expected the overhead to be significantly lower for the new approaches. In fact, the overhead (Table 3) and CPU time (Table 4) data suggest that LCFR is quite costly in comparison to the other approaches. For problems with no solution, LCFR expends the most effort before reporting a failure. All other approaches report failure as early as possible. Only in the Blocks World problems does LCFR out-perform the other approaches.

	UCPOP	LCFR	Temp.	Reuse	Prob	Bang
A	15.6	7.8	46.4	49.5	37.1	11.3
B	33.6	106.8	48.1	11.9	13.0	1.9
C	47.4	56.2	34.0	6.4	2.0	2.0
D	81.8	2021.4	89.3	90.2	89.4	31.4
E	0.8	8.8	8.1	4.9	6.4	14.4
F	37.8	556.4	97.0	54.2	41.0	16.0
G	43.6	30.4	98.6	305.5	289.7	30.4
H	181.2	2975.1	967.7	567.2	956.2	73.5
I	3.0	2.2	1.6	2.2	1.6	10.7
J	43.6	363.4	58.8	75.4	56.6	14.6

Table 4: Average CPU time in seconds, all problems.

In terms of overhead, the performance of the probabilistic reuse scheme is usually lower or comparable to the approaches other than Bang. This implies that if proper criteria, mostly likely domain and problem dependent, for reuse can be determined then the search space can be reduced greatly.

Bang-UCPOP incurs no overhead; its CPU time is the minimum in all but three domains. However, it appears to be problem dependent, rather than specifically domain dependent and so should be applied based on the type of problem rather than applying it for every problem in the domain. The primary cost of Bang-UCPOP is the storage of certain nodes to allow back tracking. If the unsuitability of certain plans can be detected very early, the search space explosion to support backtracking can be controlled.

Our template scheme assumes that the order in which similar open conditions are selected for repair does not matter. We tested this assumption by running experiments in which flaw selection from a template is randomized. The results showed no significant difference between open conditions selected randomly versus simply taking the first flaw from the template.

7 Conclusion

Not too surprisingly, no one approach seems to be best, solving all possible problems as efficiently as possible. Each solution seems to have its pros and cons, favoring some domain or problem within a domain. Though LCFR is able to solve many problems with far fewer plans examined than UCPOP, the cost of doing so, in terms of overhead, can be quite high. The four approaches described in this paper solved more problems than UCPOP, almost as many problems as did LCFR, and usually incurred far less overhead than LCFR. Additionally, the results of Bang-UCPOP suggest that flaw selection alone is not adequate for efficient planning.

However, these approaches and this comparison are barely a first step. We need to model *why* different approaches work better in different domains and problems. Such models will help determine which approaches to

	A	B	C	D	E	F	G	H	I	J
UCPOP	854	1013	4088	274	**98**	**17**	448	-	285	**101**
LCFR	69	700	310	197	104	23	**63**	761	**32**	112
Temp.	1784	784	1096	270	533	76	271	-	63	120
Reuse	**42**	598	201	270	296	506	8952	**583**	64	154
Prob	1346	589	**68**	270	386	506	-	928	64	159
Bang	2880	**549**	321	**180**	1919	1050	208	-	1065	506

Table 2: Number of plans examined, successful problems only, average results.

apply in which situations and to design new methods. For example, from the execution traces of UCPOP, we observed that reordering sub-goals or operators in the domain strongly affects the amount of search required to solve problems; in particular, some orderings lead quickly to a solution while others appear to cycle. A flaw selection strategy partly eliminates this problem, but at great expense. If we can identify what plans or orderings will lead to cycles, then we can modify plan refinement to prune those plans early in the planning process.

The two limited reuse approaches performed well on problems with related sub-goals. One simple improvement to probabilistic reuse could be to make the probability a function of number of flaws in the plan with reused steps. For example, if the number of threats introduced by applying reuse is more than that introduced by solving the minimum cost flaw, the probability of reuse should be reduced. A better way is to use more knowledge about the domain and problem to decide on step reuse rather than applying reuse with some probability. We should be able to identify long sequences (sub-plans) and solve similar flaws together rather then considering them separately. For example, in Truckworld, when the truck tries to pick up fuel drums to fill its fuel tank, it can pick up other objects it needs since the sequence of steps are same.

Considering the time reported to solve even a simple problem, the problem of scaling up to larger problems is daunting. Based on this small exploration of methods for improving plan generation efficiency, we need additional methods for constraining the search space in partial order planning and language constructs to incorporate known constraints. Most importantly, we need to know how domain dependent problem characteristics lead to inefficient exploration of the search space.

References

[Anderson *et al.*, 1993] S.D. Anderson, A. Carlson, D.L. Westbrook, D.M. Hart, and P.R. Cohen. CLASP/CLIP Common Lisp Analytical Statistics Package/Common Lisp Instrumentation Package. Technical Report TR 93-55, Computer Science Department, University of Massachusetts, 1993.

[Hanks *et al.*, 1993] S. Hanks, D. Nguyen, and C. Thomas. A beginner's guide to the Truckworld simulator. Dept of CS&E UW-CSE-TR 93-06-09, University of Washington, June 1993.

[Howe and Cohen, 1994] A.E. Howe and P.R. Cohen. Detecting and explaining dependencies in execution traces. In P. Cheeseman and R.W. Oldford, editors, *Selecting Models from Data; Artificial Intelligence and Statistics IV*, pages 71–78. Springer-Verlag, NY, 1994.

[Joslin and Pollack, 1994] D. Joslin and M. Pollack. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proceedings of the Twelth National Conference on Artificial Intelligence*, pages 1004–1009, Seattle, WA, August 1994.

[Kambhampati and Hendler, 1992] S. Kambhampati and J.A. Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence Journal*, 55(2-3), 1992.

[Penberthy and Weld, 1992] J.S. Penberthy and D.S. Weld. UCPOP: a sound, complete, partial order planner for adl. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, pages 103–114, 1992.

[Peot and Smith, 1993] M.A. Peot and D.E. Smith. Threat-removal strategies for partial-order planning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 492–499, 1993.

[Srinivasan and Howe, 1995] R. Srinivasan and A.E. Howe. New methods for plan selection and refinement in a partial-order planner. Computer Science Department 95-03, Colorado State University, 1995.

[Tate *et al.*, 1994] A. Tate, B. Drabble, and J. Dalton. The use of condition types to restrict search in an ai planner. In *Proceedings of the Twelth National Conference on Artificial Intelligence*, pages 1129–1134, Seattle, WA, 1994.

[Wilkins, 1988] D.E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1988.