Plan-Refinement Strategies and Search-Space Size[1]

Reiko Tsuneto reiko@cs.umd.edu
Dana Nau nau@cs.umd.edu
James Hendler hendler@cs.umd.edu

Department of Computer Science
and Institute for Systems Research
University of Maryland
College Park, MD 20742
USA

Abstract

During the planning process, a planner may have many options for refinements to perform on the plan being developed. The planner's efficiency depends on how it chooses which refinement to do next. Recent studies have shown that several versions of the popular "least commitment" plan refinement strategy are often outperformed by a *fewest alternatives first* (FAF) strategy that chooses to refine the plan element that has the smallest number of alternative refinement options.

In this paper, we examine the FAF strategy in more detail, to try to gain a better understanding of how well it performs and why. We present the following results:

- A refinement planner's search space is an AND/OR graph, and the planner "serializes" this graph by mapping it into an equivalent state-space graph. Different plan refinement strategies produce different serializations of the AND/OR graph.
- The sizes of different serializations of the AND/OR graph can differ by an exponential amount. A planner whose refinement strategy produces a small serialization is likely to be more efficient than a planner whose refinement strategy produces a large serialization.
- The FAF heuristic can be computed in constant time, and in our experimental studies it usually produced an optimal or near-optimal serialization. This suggests that using FAF (or some similar heuristic) is preferable to trying to guarantee an optimal serialization (which we conjecture is a computationally intractible problem).

Keywords: planning and search; refinement strategies; commitment strategies

---

[1] This research was supported in part by grants from NSF (IRI-9306580 and EEC 94-02384), ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), the ARPA/Rome Laboratory Planning Initiative (F30602-93-C-0039), the ARPA I3 Initiative (N00014-94-10907), the ARL (DAAH049610297), and ARPA contract DABT-95-C0037. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the funders.

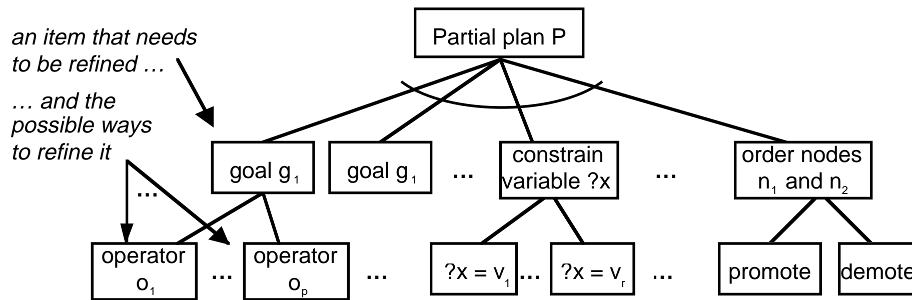


Figure 1. Possible refinement choices in planning.

1. Introduction

One characteristic of partial-order planners—regardless of whether they are Hierarchical Task Network (HTN) planners such as UMCP [Erol, 1995] and O-Plan [Currie and Tate, 1994], or planners that use STRIPS-style operators such as UCPOP [Penberthy and Weld, 1992]—is that they search a space in which the nodes are partially developed plans. The planner refines the plans into more and more specific plans, until either a completely developed solution is found or every plan is found incapable of solving the problem.

During this process, a planner may often have many different options for what kind of refinement to perform next, as illustrated in Figure 1. A planner that uses STRIPS-style operators may need to choose which unachieved goal to work on next, which operator to use to achieve a goal, or which technique to use (promotion, demotion, or variable separation) to resolve a goal conflict. An HTN planner usually has an even larger array of options: it may need to choose which unachieved task to work on next, which method to use to accomplish the task, or which constraint (from among a number of different possibilities) to impose on the plan. The planner's efficiency depends greatly on its *plan refinement strategy*, which is the way it goes about choosing among these options.

In the planning literature, the term "least commitment" generally refers to a refinement strategy in which the planner postpones making some particular kind of refinement until it is forced to do so. For example, if a planner uses a "least commitment to step orderings" strategy, then whenever more than one ordering is possible among the steps of a plan, the planner will avoid committing to a particular ordering unless it must do so in order to proceed with the rest of the planning. The "least commitment" idea was originally applied to step orderings [Sacerdoti, 1975], but it has also been applied to other kinds of refinements. For example, Stefik's MOLGEN program [Stefik, 1981] used a "least commitment to constraint posting" approach; and Tsuneto *et al.* [1996] have examined both a "least commitment to variable bindings" strategy and a "least commitment to task achievement" strategy for HTN planning.

One reason why least-commitment strategies are useful is that if the planner can avoid making refinements prematurely, this can reduce the number of alternative plans it might need to examine. However, it is not necessarily a good idea to apply the same least-commitment strategy throughout the entire planning process. In order

to do planning at all, a planner has to refine *something*[2]—and thus, when a planner postpones refining one aspect of the plan it is generating, this may make it prematurely refine some other aspect of the plan. This suggests that it may be better to choose dynamically among different kinds of refinements throughout the planning process.

One way to choose what kind of refinement to make next is to look at all of the items that need to be refined in the current partial plan, and choose whichever one has the fewest number of alternative possible refinements. Two versions of this "*fewest alternatives first*" heuristic have been examined in the AI planning literature. For partial-order planning with STRIPS-style operators, Joslin and Pollack [1994; 1996] found that a version of this strategy outperformed the "least commitment to step orderings" strategy; and for HTN planning, Tsuneto *et al.* [1996] found that a version of this strategy outperformed both a "least commitment to variable bindings" strategy and a "least commitment to task achievement" strategy.

Although AI planning researchers have begun to investigate the FAF heuristic only recently, a similar heuristic has been known in the constraint satisfaction literature for more than 20 years: Bitner and Reingold [1975] used it as a search rearrangement method, and Purdom [1983] analyzed its application to SAT problems. One reason why it has taken so long for this heuristic to become known to AI planning researchers is that the relationship between control strategies for search algorithms and refinement strategies for AI planning is a rather complicated one, whose precise nature has not been clearly understood. In this paper, we examine that relationship in detail, and present the following results:

- The search process that is carried out by an AI planning system corresponds to taking an AND/OR graph and generating from it an equivalent state-space graph, one OR-branch at a time. This process we call *serializing* the AND/OR graph. Different plan refinement strategies produce different serializations of the AND/OR graph.
- Different serializations of the AND/OR graph contain different numbers of nodes, and the largest serialization can contain an exponentially greater number nodes than the smallest one. In the worst case, the planner may need to examine every node in the search space—so a planner whose search space is small is likely to be more efficient than a planner whose search space is large.
- The FAF strategy uses a greedy heuristic: each time it needs to decide which OR-branch to include next in the search space, it chooses the one that has the smallest number of branches. This heuristic does not always result in the smallest possible serialization—but it can be computed in constant time, and in our studies it usually resulted in a serialization that was either optimal or near-optimal.
- We conjecture that when the planner needs to decide which OR-branch to include next in the search space, the task of deciding which OR-branch is *optimal* (i.e., which OR-branch is in the smallest possible serialization) is an NP-hard problem. If this conjecture is correct, this suggests that any plan refinement strategy that is guaranteed to produce the smallest possible search space will incur an unacceptably high overhead—and thus it is better for AI planning systems to use

[2] Of course, this refinement need not necessarily be an irrevocable one. Most modern planning systems use a tentative control strategy such as backtracking, so that they can go back and undo decisions that do not work out.

a refinement strategy such as FAF which is quickly computable and gives good results most of the time.

2. Partial-Order Planning and AND/OR Graphs

As illustrated in Figure 1, the space searched by a partial-order planner may be thought of as an AND/OR graph in the following manner:

- In a partially developed plan, there may be several elements of the plan that need to be refined in one way or another. These may include both unachieved goals or tasks (which would be refined by finding ways to achieve them), and unsatisfied constraints (which might be satisfied by binding variables or specifying node orderings). All of these elements will sooner or later need to be refined—and thus the choice of which refinement to make next corresponds to an AND-branch in the planner's search space.
- For each element that needs refining, there may be more than one way to refine it (for example, several ways to instantiate a variable, or several operators or methods applicable to an unachieved goal or task), generating different partial plans. Any applicable refinement will be satisfactory provided that it produces a satisfactory plan—and thus the choice of how to reduce an element corresponds to an OR-branch in the planner's search space.

If the refinements performed on a plan were independent in their effects on the plan, a refinement planner could search the AND/OR graph directly, building up a solution to the planning problem straightforwardly by finding independent solutions to subproblems and composing them into solutions to higher-level problems. However, since the goals usually are not independent, refinement planners usually do not decompose the search space. Instead, when they refine some element of a plan, they keep track not only of the element that is being refined, but also of the entire rest of the plan. Thus, the planner searches a state-space graph that is a "serialization" of the AND/OR graph.

Although the concept of serializing an AND/OR graph is conceptually straightforward, the formal definition is rather complicated notationally. To keep the notation simple, in this paper we give a formal definition only for the special case where the AND/OR graph is binary (i.e., each non-leaf node has exactly two children). We trust that it will be obvious to the reader how to generalize this definition for the case where the AND/OR graph is not binary.

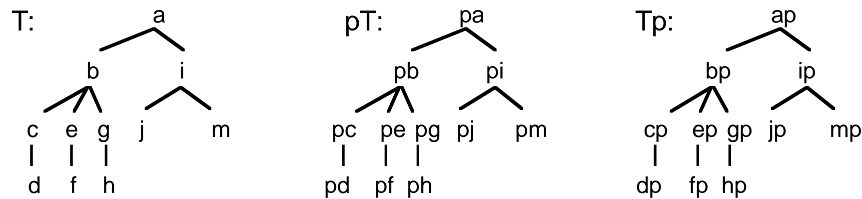


Figure 2. A tree T, and the trees pT and Tp (where p is a node not in T).

First, we will need the following notation (see Figure 2 for examples). Let T be a tree whose node set is N and whose edge set is E. If p is any node not in N, then:

- Tp is the tree with node set $\{np : n \in N\}$ and edge set $\{(mp,np) : (m,n) \in E\}$;

- pT is the tree with node set $\{pn : n \in N\}$ and edge set $\{(pm,pn) : (m,n) \in E\}$.

If G is a binary AND/OR graph, there are three possible cases for what its serializations are:

Case 1: G consists of a single node. Then the only serialization of G is G itself.

Case 2: G contains more than one node, and the branch at G's root node g is a binary OR-branch leading to two subgraphs H and I. If S and T are serializations of H and I, respectively, then as shown in Figure 3, the tree R whose root is g and whose subtrees are S and T is a serialization of G.

Case 3: G contains more than one node, and the branch at G's root g is a binary AND-branch leading to two subgraphs H and I. Let S and T be serializations of H and I, respectively. Let S's root be s and its leaf nodes be $s_1, s_2, \ldots, s_p$; and let T's root be t and its leaf nodes be $t_1, t_2, \ldots, t_q$. Then as shown in Figure 4, the following trees are serializations of G:

- the tree R_1 formed by taking the tree St, and attaching to its leaves $s_1t, s_2t, \ldots, s_pt$ the trees $s_1T, s_2T, \ldots, s_pT$, respectively;
- the tree R_2 formed by taking the tree sT, and attaching to its leaves $st_1, st_2, \ldots, st_q$ the trees $St_1, St_2, \ldots, St_q$, respectively.

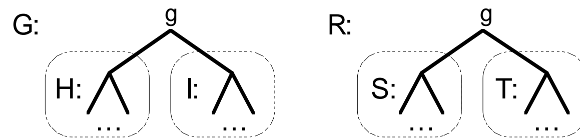


Figure 3. Case 2 of serializing an AND/OR graph.

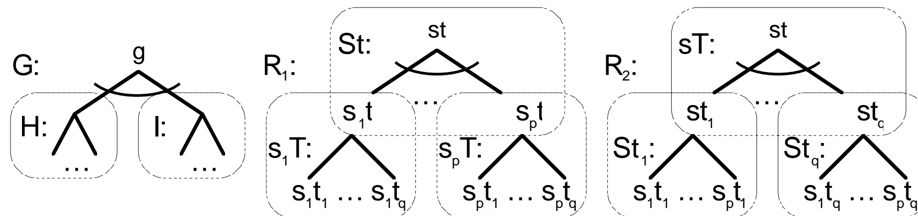


Figure 4. Case 3 of serializing an AND/OR graph.

In Figure 4, both serializations of the AND/OR graph have the same number of nodes—but this needs not always be the case. As an example, Figure 5 shows another AND/OR graph, and three possible serializations of it. Note that in each serialization, the set of leaf nodes is exactly the same. Furthermore, for each leaf node, the number of paths—and the set of operations along each corresponding path—are also the same. What differs is the order in which these operations are performed—and since different operations produce different numbers of children, this means that different serializations contain different numbers of nodes.

The idea of serializing an AND/OR graph occurs in a number of search procedures, although the first case we know of where such a technique was described explicitly was in the SSS* game-tree search procedure [Stockman, 1979]. One well

known example is Prolog's search procedure (for example, see [Clocksin and Mellish, 1981], which serializes AND/OR graphs in a depth-first left-to-right manner. For example, in the graph G of Figure 5, suppose that each node corresponds to a logical atom, each AND-branch corresponds to a Horn clause, and each OR-branch corresponds to the different ways a literal might match the head of a Horn clause. Then Prolog would do a depth-first search of the tree S_1. In general, the number of possible serializations of an AND/OR graph can be combinatorially large; for example, there are ten possible serializations of the graph G of Figure 5. Which serialization will actually be used depends on the search procedure. For example, a procedure that achieves goals and subgoals in a depth-first left-to-right fashion (as Prolog does) would serialize G into S_1, but a procedure that achieves goals and subgoals in a depth-first right-to-left fashion would serialize G into S_3 instead.

Obviously, a planner will not necessarily examine every node in its serialized search tree. It may prune some of these nodes as infeasible, and it may find its desired solution before it examines all of the unpruned nodes. However, in the worst case, the planner will need to examine every one of the nodes in the serialized search tree. In such a case, a planner that searches the tree S_3 of Figure 5 will be more efficient than a planner that searches the trees S_1 or S_2.

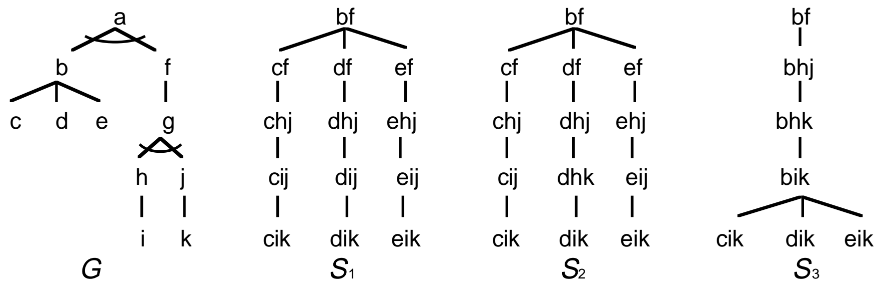


Figure 5. A simple AND/OR graph G, and three serializations S_1, S_2, and S_3.

4. Best and Worst Serializations

If we could find a serialization strategy that would always find the smallest serialization of an AND/OR graph, how much would this help? To get an idea of the answer, suppose we take the pattern shown in Figure 6a, and use it repeatedly to form an AND/OR tree $G_{b,k}$ of height $2k$, as shown in Figure 6b. In $G_{b,k}$, the number of occurrences of the pattern is

$$c_{b,k} = 1+(\mathrm{b}+1)+(b+1)^2+\ldots+(b+1)^{k-1} = \Theta(b^k),$$

so the total number of nodes in $G_{b,k}$ is

$$n(G_{b,k}) = 1+(b+3)c_{b,k} = \Theta(b^k).$$

Let $T^-_{b,k}$ and $T^+_{b,k}$ be the serializations of $G_{b,k}$ that have the smallest and largest node counts, respectively. Both of these trees have the same height, which can be calculated recursively as follows:

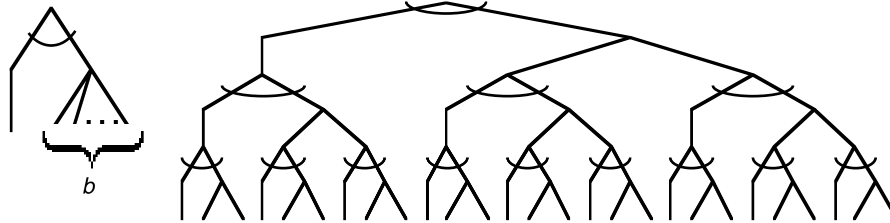
$$h(T^-_{b,k}) = h(T^+_{b,k}) \quad = \quad \begin{cases} 2 & \text{if } k = 1, \\ 2h(T^+_{b,k-1}) + 2 & \text{otherwise} \end{cases}$$
$$= \quad \sum_{i=1}^{k} 2^i \quad = \quad 2^{k+1} - 2.$$

$T^-_{b,k}$ and $T^+_{b,k}$ both consist of 2^k-1 levels of unary OR-branches interspersed with 2^k-1 levels of b-ary OR-branches. However, $T^-_{b,k}$ has its unary OR-branches as near the top as possible and its b-ary OR-branches as near the bottom as possible; and vice versa for $T^+_{b,k}$. As shown in Figure 6c, the branches at the top k levels of $T^-_{b,k}$ are all unary, and those at its bottom 2^{k-1} levels are all b-ary; the reverse is true for $T^+_{b,k}$.

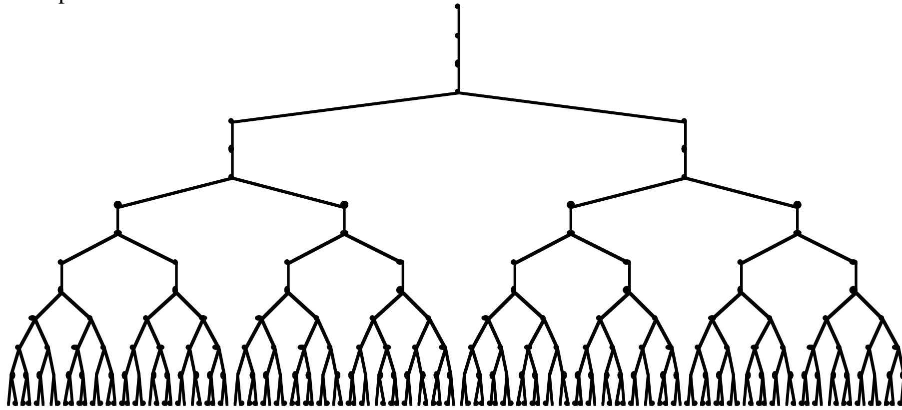
Calculating the node counts for $T^-_{b,k}$ and $T^+_{b,k}$ is too complicated to do here, but in a forthcoming technical report we show that the numbers of nodes in these trees are $n(T^-_{b,k}) = \Theta(b^{2^k})$ and $n(T^+_{b,k}) = \Theta(2^k b^{2^k})$. Thus, the numbers of nodes in the worst possible serialization and the best possible serialization differ by a multiplicative factor of $\Theta(2^k)$.

b

(a) Basic pattern, with parameter b.



(b) AND/OR tree $G_{2,3}$ produced by the pattern if $b = 2$ and $k = 3$.



(c) The smallest possible serialization $T^-_{2,3}$ of $G_{2,3}$.

Figure 6. An AND/OR tree formed by repetitions of a pattern; and the smallest possible serialization of the AND/OR tree.

4. Fewest Alternatives First

During the course of its operation, an AI planning algorithm will generate a serialization of an AND/OR graph one OR-branch at a time. For example, starting

from the node *a* in the AND/OR graph *G* shown in Figure 5, the first choice is whether to expand the OR-branch rooted at *b* or the OR-branch rooted at *f*. If we choose *b* then we will end up with a search space similar to S_1 or S_2; and if we choose *f* then we will end up with a search space similar to S_3. One way to choose which OR-branch to expand next is to use the "fewest alternatives first" (FAF) heuristic of Section 1. In many cases, this simple heuristic produces optimal results. For example, in Figure 5, this heuristic would choose to expand *f*, *h*, and *j* before expanding *b*, thereby producing the tree S_3.

FAF also is easy to compute. The cost of computing FAF at any node *n* is $O(c(n)+g(n))$, where $c(n)$ is the number of *n*'s children, and $g(n)$ is the number of *n*'s grandchildren. Thus, if one assumes (as is typical in analyses of AI search algorithms) that the branching factor of each node is bounded by some constant *b*, then the cost of computing FAF is $O(b + b^2) = O(1)$.

In empirical studies on various planning domains, adaptations of the FAF strategy have performed quite well in comparison with other popular refinement strategies. The "least cost flaw repair" strategy investigated by Joslin and Pollack [1994, 1996] uses the FAF heuristic to choose among all of the refinements available to a STRIPS-style planner; and the "DVCS" strategy investigated by Tsuneto *et al.* [1996] for HTN planning uses the FAF heuristic to choose among some (but not all) of the refinements available to an HTN planner. In these studies, least-cost flaw repair and DVCS outperformed a number of other strategies, including the well known "least commitment to step orderings" strategy.

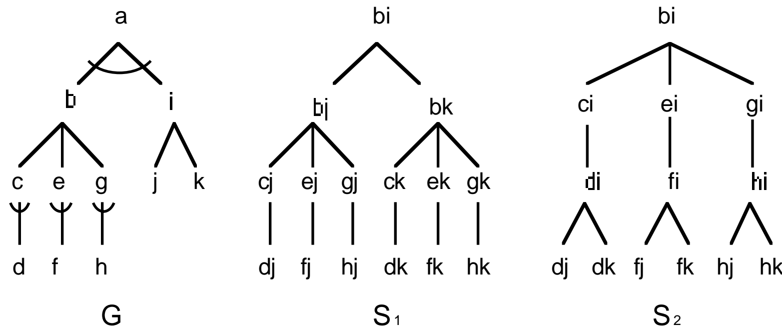


Figure 7. A situation where the FAF heuristic fails to produce the best serialization of an AND/OR graph. FAF chooses to expand *i* before *b*, thus producing S_1; but S_2 contains fewer nodes.

Despite its good empirical performance, FAF does not always produce optimal results. For example, consider the graph *G* of Figure 7. To serialize *G*, FAF would choose to expand *i* before expanding *b*, thus producing the tree S_1. However, if it had chosen to expand *b* first, it would have been able to produce the smaller tree S_2. This situation is reminiscent of what happens in a number of NP-hard optimization problems, in which the obvious greedy heuristics will make the best choice in a large number of situations, but will sometimes make choices that cause greater costs to be incurred later on. At least one example of this occurs in the AI planning literature, involving a greedy heuristic for the block-stacking problem [Gupta and Nau, 1992]

To formalize the notion of "optimal results" in the previous paragraph, first we define a *minimal serialization* of an AND/OR graph *G* to be a serialization *T* of *G*

such that no other serialization of G contains fewer nodes than T. Now, suppose we have an AND/OR graph G whose root branch is an AND branch. Let the children of the root node be $n_1, \ldots, n_k$. Then for $i = 1,\ldots,k$, the node n_i is an *optimal candidate* for expansion if there is a minimal serialization T of G whose root branch is formed by expanding n_i. For example, f is the optimal candidate in Figure 5, and b is the optimal candidate in Figure 7. We conjecture that finding an optimal candidate for expansion is NP-hard.

5. Experimental Studies of FAF

As discussed above, FAF usually seems to do better than other popular heuristics, but can sometimes do poorly. This raises two important questions: how close FAF comes (on the average) to finding the best possible serialization, and how it compares (on the average) with the best, worst, and/or average serializations. We have begun an experimental exploration to try to answer these questions.

We have compared the performance of FAF with an average serialization performed on 50 different randomly generated AND/OR trees. The sample trees were generated using a tree generation algorithm based on [Luke, 1997]. These trees had 1 to 5 branches at each node, with a maximum depth of 8. All nodes at even depths were AND-nodes, while all nodes at odd depths were OR-nodes. Thus, leaves were only placed at even depths. The algorithm was set to generate 50 random trees with an average close to 30 and the average depth close to the maximum. In the population that was actually generated, the average tree size (number of nodes) is 32.32 and the average depth is 7.64. The smallest tree is of size 19 and the largest tree is of size 51. The number of serializations for the trees varied from 1 through to over half a million.

To find the best and worst serializations, we developed a program to exhaustively enumerate all serializations, keeping track of minimum, maximum and average size. Due to the extreme number of serializations for many of the trees, we ran this program for up to 50,000 serializations. If the first program had not enumerated all serializations by this cutoff (i.e. there were more 50,000 serializations for the given tree), we instead used a separate algorithm which randomly generated 50,000 trials.[3] The minimum, maximum and average were again collected.

The FAF algorithm was also run on each tree, by applying the heuristic at each AND-node expansion (when there were more than two smallest branches, the leftmost one was chosen). Data on the number of serializations, minimum, maximum, average, and FAF sizes are all shown in Table 1. The large variance in the sizes of the serializations makes comparison of the raw data difficult. It is easy to see, however, that in 32 of the 47 cases where there was more than a single serialization size, the FAF algorithm found the optimal solution.

To see how the algorithm performs overall, and to compare the algorithm to the averages, we needed a means to measure performance. In Figure 8—which shows the performance of FAF versus the average—we normalized the results, with 0 representing the best overall serialization and 1 representing the worst. In 46 of the 47 cases, FAF performed better than the average. 32 times the optimal was found, and 44 further times the algorithm performed better than half way between optimal and average. We believe these results are quite encouraging, showing that the FAF algorithm performs quite well in the average case.

---

[3] These 50,000 could not be guaranteed unique without prohibitive computational costs; however a very large sample population was probabilistically guaranteed.

Table 1. Experimental results.

Tree	Number of possible serializations	Size of smallest serialization	Size of largest serialization	Average serialization size	Size of serialization found by FAF
1	4228	38	64	43.6	39
2	4	33	34	33.2	33
3	>50000	156	275	185.0	154
4	352	64	73	66.3	64
5	>50000	123	204	143.6	121
6	>50000	71	126	86.0	71
7	7	18	25	20.0	18
8	56	19	30	22.1	19
9	>50000	259	321	277.6	259
10	17424	156	175	162.0	159
11	5284	61	67	62.9	61
12	>50000	1488	2170	1697.0	1450
13	>50000	267	340	292.7	267
14	>50000	253	463	331.0	255
15	>50000	229	274	246.8	235
16	>50000	158	254	196.6	158
17	>50000	744	861	791.5	746
18	16777	56	93	71.2	56
19	180	29	44	35.4	29
20	>50000	109	157	129.8	111
21	4	36	38	36.8	36
22	>50000	117	136	125.4	117
23	14	17	23	19.6	17
24	>50000	84	122	101.2	84
25	5792	49	56	52.1	49
26	>50000	334	434	374.0	322
27	146	40	49	44.2	40
28	100	106	115	110.2	108
29	8992	71	83	76.6	71
30	44	32	41	36.4	34
31	>50000	335	434	381.4	330
32	4	33	34	33.5	33
33	3	28	29	28.5	28
34	20	27	33	30.0	28
35	>50000	354	462	405.9	348
36	>50000	162	184	173.2	165
37	28	40	45	42.5	40
38	>50000	226	327	280.9	239
39	>50000	249	310	282.4	278
40	>50000	173	225	201.5	173
41	>50000	237	355	300.6	232
42	>50000	659	929	803.1	643
43	>50000	80	86	83.5	83
44	>50000	520	621	580.2	525
45	60	27	36	32.7	27
46	>50000	161	226	207.6	161
47	1014	70	82	79.6	82
48	1	15	15	15.0	15
49	2	17	17	17.0	17
50	4	24	24	24.0	24

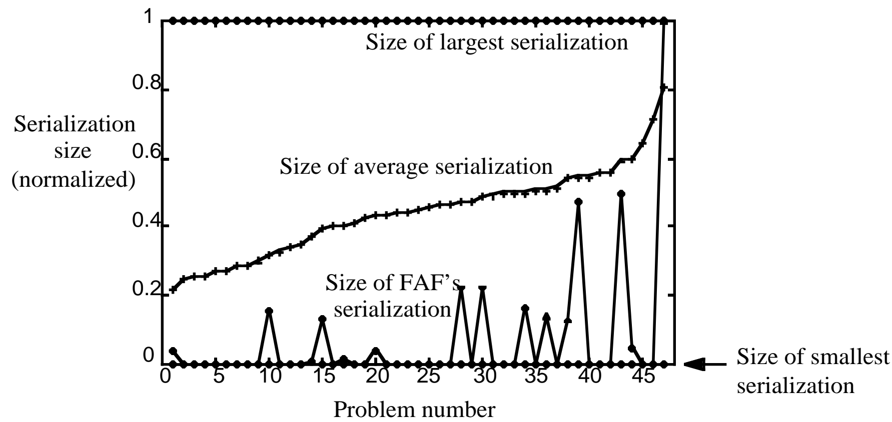


Figure 8. Sizes of FAF(◇) and average(+) serializations normalized with the smallest and largest serialization sizes.

There was, however, one case in which FAF produced the worst serialization (this is also the only case where FAF did worse than average). To see the cause of this, we analyzed the particular tree (shown in Figure 9). In this case, FAF could have produced the best serialization if it had chosen the right child of the root to expand first instead of the left child. Since our program simply chose leftmost in the case of the tie, FAF did poorly in our test. This does, however, show a potential weakness in implementations of FAF for planning, since it have an additional heuristic for use in this case. Examining what to do in this case could lead to further improvement of planning choice mechanisms, and this is a topic we are currently exploring.

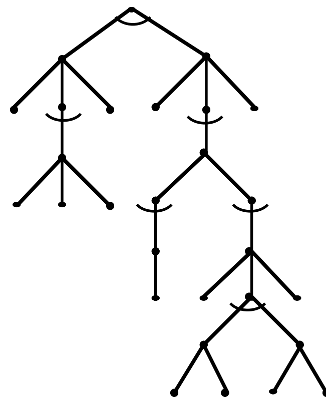


Figure 9. The one tree in which FAF produced the worst serialization.

6. Conclusions

In this paper, we have studied a plan refinement strategy, the "fewest alternatives first" (FAF) strategy, that chooses among various kinds of refinements depending on which one has the smallest number of alternative choices. In recent studies by Joslin and Pollack [1994, 1996] and Tsuneto *et al.* [1996], FAF usually outperformed

several different “least commitment” refinement strategies. In this paper, we have examined the FAF strategy in more detail in an attempt to understand how well it performs and why.

We have shown that the search process that is carried out by an AI planning system corresponds to “serializing” an AND/OR graph—mapping it into an equivalent state-space graph. Different plan refinement strategies thus correspond to different ways to serialize the AND/OR graph representing the planning choice points.

Different serializations of an AND/OR graph have different sizes, and the smallest serialization can be exponentially smaller than the largest one. We have shown that a planner whose plan refinement strategy produces a small serialization of the AND/OR graph is likely to be more efficient than a planner whose plan refinement strategy produces a large serialization.

Like most greedy heuristics, FAF does not always produce optimal results—but in our studies it usually produced a serialization that was either optimal or near-optimal. If our conjecture is correct that any strategy that guaranteed the smallest possible serialization would be intractable to compute, then this suggests that it is better for AI planning systems to use a plan refinement strategy such as FAF, that is quickly computable and usually gives good results.

We believe these results explain why FAF performs well in the previous studies, and opens several interesting issues for exploration. First, as we have noted, better serializations produce smaller search spaces, thus potentially improving planning behavior. However, the exact relationship between a given planner and this search space is quite complex, and there may be cases where certain planners interact better with certain serializations. Second, while FAF performs quite well, it is clear that there is still plenty of room for improvement. This can include looking for algorithms that can better optimize search space (serialization) size, improvements on FAF (for example better tie-breaking rules), and identification of analytic techniques that could analyze the tree formed by the operators and better select or prune the search spaces.

Finally, we are beginning to explore other effects of search control on planning performance. Given the correspondence between these discussed in the paper, it seems that other ways of controlling search, particularly pruning unpromising branches, may also be successful. Gerevini and Schubert [1996] showed that various pruning strategies have beneficial properties for the UCPOP planner, and we plan to extend this work, examining how pruning can effect search in serializations of planning trees as discussed in the paper.

References

[Barret and Weld, 1994] Anthony Barret and Daniel Weld. Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence* 67(1), pp. 71–112.

[Bitner and Reingold, 1975] James Bitner and Edward Reingold. Backtrack Programming Techniques. *CACM* 18(11), pp. 651–-656.

[Clocksin and Mellish, 1981] W. Clocksin and C. Mellish. *Programming in PROLOG.* Springer-Verlag.

[Currie and Tate, 1991] Ken Currie and Austin Tate. O-plan: the open planning architecture. *Artificial Intelligence* 52, pp. 49–86.

[Erol, 1995] Kutluhan Erol. *HTN planning: Formalization, analysis, and implementation.* Ph.D. dissertation, Computer Science Dept., U. of Maryland.

[Gupta and Nau, 1992] Naresh Gupta and Dana Nau. On the complexity of blocks-world planning. *Artificial Intelligence* 56:2-3, pp. 223–254.

[Gerevini and Schubert, 1996] Alfonso Gerevini and Lenhart Schubert. Accelerating Partial-Order Planners: Some Techniques for Effective Search Control and Pruning. *Journal of Artificial Intelligence Research* 5, pp. 95– 137.

[Joslin and Pollack, 1994] David Joslin and Martha Pollack. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 1004– 1009.

[Joslin and Pollack, 1996] David Joslin and Martha Pollack. Is "early commitment" in plan generation ever a good idea? In *Proc. Thirteenth National Conference on Artificial Intelligence*, pp. 1188-1193.

[Kambhampati *et al.*, 1995] Subbarao Kambhampati, Craig Knoblock, and Qiang Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence* 76, pp. 167–238.

[Kumar, 1992] Vipin Kumar. Algorithms for constraint -satisfaction problems: A survey. *AI Magazine*, pp. 32–44.

[Luke, 1997] Sean Luke. A Fast Probabilistic Tree Generation Algorithm. Unpublished manuscript.

[Penberthy and Weld, 1992] J. S. Penberthy and Daniel Weld. UCPOP: A sound, complete, partial order planner for ADL. *Proc. KR-92*.

[Purdom, 1983] Paul W. Purdom. Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence* 21, pp. 117– 133.

[Purdom and Brown, 1983] Paul W. Purdom and Cynthia A. Brown. An Analysis of Backtracking with Search Rearragement. *SIAM J. Computing* 12(4), pp.717– 733.

[Sacerdoti, 1977] Earl Sacerdoti. *A Structure for Plans and Behavior.* American Elsevier Publishing Company.

[Stefik, 1981] Mark Stefik. Planning with constraints (MOLGEN: part 1). *Artificial Intelligence* 16, pp. 111–140.

[Smith *et al.,* 1996] S. J. J. Smith, D. S. Nau, and T. A. Throop. Total-order multi-agent task-network planning for control bridge. *AAAI-96*, pp.108-113.

[Stockman, 1979] G. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence* 12(2), pp. 179–96.

[Tsuneto *et al*., 1996] Reiko Tsuneto, Kutluhan Erol, James Hendler, and Dana Nau. Commitment strategies in hierarchical task network planning. In *Proc. Thirteenth National Conference on Artificial Intelligence*, pp. 536-542.

[Veloso and Stone, 1995] Manuela Veloso and Peter Stone. FLECS: Planning with a flexible commitment strategy. *JAIR* 3, pp. 25–52.