ABSTRACT

Title of Dissertation: Efficient Refinement Strategies for HTN Planning

Reiko Tsuneto, Doctor of Philosophy, 1999

Dissertation directed by: Professor James Hendler
Department of Computer Science

Professor Dana Nau
Department of Computer Science

The efficiency of planning greatly depends on the type of refinement strategy the planner uses, but little is understood about the conditions under which each refinement strategy performs well. This is especially true of refinement strategies for hierarchical task network (HTN) planning, even though many real-world planning applications use the HTN planning technique.

This dissertation presents and analyzes three heuristics for refinement strategies for HTN planning. Through these analyses, I have produced the following results:

- **Least-commitment versus FAF.** There are some planning domains where the least commitment strategy, the most popular strategy in refinement planning, does not perform well. However, the problem can be resolved in many

cases by applying a different heuristic called the fewest alternative first (FAF) heuristic.

- **Refinement as AND/OR tree search.** Different refinement strategies can be viewed as different ways to transform an AND/OR graph into an OR-tree. The efficiency of a refinement strategy can then be evaluated by taking the size of the OR-tree the strategy generates. In an empirical analysis, FAF generated the smallest OR-tree possible by this evaluation method.

- **The ExCon heuristic.** This refinement heuristic improves the efficiency of planning by detecting and handling possibly problematic interactions between tasks. In a comparison of this heuristic against the FAF heuristic, ExCon performed increasingly better than FAF on problems where there are many task interactions.

- **Analysis of the effects of ordering constraints.** The ordering constraints in HTN planning domains can make an impact on the planning efficiency. In particular, a planner can use the left-to-right (LtoR) heuristic, which plans in a way similar to a forward planner, to plan efficiently on problems where there are many ordering constraints. Also, empirical tests show how many ordering constraints are necessary for LtoR to perform well. Furthermore, LtoR can be combined with ExCon to take advantage of both heuristics. The resulting strategy performed well on most of the test domains.

Efficient Refinement Strategies for HTN Planning

by

Reiko Tsuneto

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1999

DEDICATION

To Dave for his support and enscouragement

ACKNOWLEDGEMENTS

My greatest gratitude goes to my mother, father, and sisters, who supported me in every way. Their continuous encouragement has given me the strength to go through the doctoral program.

I was so fortunate to have two great advisors, Prof. Jim Hendler and Prof. Dana Nau. I have enjoyed working with them both. I also want to thank other members of my defense committee for their guidance.

Edna Walker helped me with many processes associated with graduate research. Like every other graduate student in this department, I am grateful to Nancy Lindley, who has done a great job of making us feel a part of the department. She always had an answer when I had a question.

I want to thank previous and current members of the PLUS group and the DSN group. They taught me what real Americans are like. Kutluhan Erol was a mentor to me when I first started my research in planning. Sean has been a great help with graphs and figures, and many computer troubles I encountered. Amnon gave me helpful feedback on my practice talks.

I also want to thank my best friends I have made at the Maryland. Yahui gave me a lot of support when I was writing the thesis proposal. We had fun together on many weekends, away from our graduate work. Leliane de Barros and I had a lot of discussions about our research in planning. She cheered up our office during the year she stayed with us.

Finally, I owe so much to Dave Rager, who proofread this thesis and listened patiently to my whining. He gave me the support and encouragement when I most needed it.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

An AI planning system searches a space of partially-developed plans in order to find a solution plan that satisfies every requirement of a problem. Since the search space may be infinite, it is important to search for a solution plan efficiently. One of the most important influences on the efficiency of a planner's search is its *refinement strategy* (defined in Section 1.1).

This dissertation addresses plan refinement strategies for HTN planning, in order to understand how refinement strategies can improve the HTN planner's efficiency. I have analyzed the tradeoffs between problem characteristics and the performance of refinement strategies as well as creating and improving existing refinement strategies.

1.1 Background

Many planning systems are based on the idea of "refinement search" in which the planner gradually 'refines' partially-developed plans into more and more detailed plans, until it finds a plan that is fully detailed and consistent with the problem requirements. Such planning systems are called *refinement planners*. During the planning process, a refinement planner needs to choose what refinement operation to apply to the current partially-developed plan. Such a decision is made by the planner's 'plan refinement strategy'. Using an appropriate plan refinement strategy is essential for the efficiency of any refinement planner because it defines the way in which the planner plans.

There are currently two major refinement planning methodologies in AI planning. One is action-based planning and the other is hierarchical task-network (HTN) planning. In action-based planning, a state of the world is specified by a set of first-order logic predicates. The domain is described by a set of operators specified with preconditions and effects. A precondition of an operator is a condition that must be true in the world state for the operator to be applicable. For example, for the operator open-door(Robot, Door), a precondition might be that the Robot is at the Door and and another precondition might be that the Door is currently closed. An effect of an operator is the change in the world state as a direct result of the operator execution. For example, the same operator open-door(Robot, Door) might have an effect that Door is open and no longer closed. Given a fully specified initial state and a partially specified final (goal) state, an action-based planner generates a sequence of operators which, when sequentially applied to the initial state, achieves the goal state.

In HTN planning, a problem is given as a rough plan which consists of tasks to accomplish and goals to achieve. Each task or goal is then decomposed into smaller tasks using one of the *decomposition methods* specified in the domain description, transforming the rough plan into a more and more detailed plan. Similar to action-based planning, tasks in HTN planning have conditions and effects. When all the tasks in the plan are executable actions and consistent with the domain requirement, the planner outputs the plan as a solution to the problem. Compared with action-based planning, HTN planning is more expressive and is capable of incorporating more domain knowledge. Many practical planning applications use some type of HTN planning [1, 53, 2, 43].

1.2 Motivation

Recently, many planning systems have made successes in real-world applications. Many of these systems use the HTN planning approach, including applications for computer bridge playing [43], space-craft operations [1] and military operations [53]. Most of these real-world planners need to incorporate temporal reasoning, numerical computation and geometrical reasoning, yet need to plan in reasonable time. In order to do so, most of them use a planning framework that is not based on a solid theoretical foundation. Also, while they provide various search mechanisms, they require the user to make search control choices. Although this may be quite effective if the user needs the efficiency and flexibility it offers, it also poses the following problems:

- If the planner is not based on a sound and complete planning framework, there is no guarantee that a generated plan is correct or that the planner can find a plan if one exists. While this problem can be avoided in many cases by carefully encoding the domain, it is preferable to use a sound and complete algorithm.
- The user needs to be familiar with the particular planning system in order to specify the application domain in such a way that (a) the planner will generate desired solution plans, and (b) the planner will do so efficiently.

This puts too much burden on the user.

- The user may not know that the search control he provided is appropriate for the domain problems unless he tracks the planning process in detail. This is a time-consuming task for any application domain beyond small toy domains.

- Many real-world applications require updating. Often, this must be done by someone who did not write the original domain specification. Doing so without jeopardizing the consistency and efficiency of the domain can be difficult.

For these reasons, it is better for a planner to have the capability to plan efficiently with little or no search control input from the user while giving the user the choice to control of the search if (s)he wants.

Improving the efficiency of planning by employing better search strategies has been addressed in many literatures [4, 33, 26]. However, most of them are for action-based planning; few studies have been done to analyze the efficiency of search strategies for HTN planning. This is mostly due to a historical reason; unlike action-based planning, there had been no sound and complete HTN planning framework to test various search strategies until recently. The implementation of a general domain HTN planning system called UMCP (Universal Method Composition Planner) based on Erol's generalization of HTN planning [13] enabled us to analyze and evaluate many search strategies for various problem domains. I have implemented all the strategies on the UMCP planner in order to empirically analyze and compare them. Since all the refinement strategies presented in this paper are sound and complete as defined in Section 2.3, they preserve the soundness and completeness of UMCP.

1.3 Approach

Many papers on search strategies present new strategies and compare them with older strategies on several toy domains to show an improved performance. These performance evaluations fail to examine the tradeoffs between the domain characteristics and the performance of the particular strategies. Also, many of these comparisons conclude with a statement "strategy A is better than strategy B", even though this is not true in all cases. Furthermore, most of these evaluations lack statistical evaluations of the results and thus it is not clear whether or not the new strategy is significantly better than other strategies with a high confidence level. A more preferable way to compare strategies is with more controlled experiments such as those shown by Kambhampati, et al [23], where they compared different refinement planning strategies on an artificial domain. In those experiments they tested different planning strategies on problems where they could vary several factors in order to show the relationships between the performance of certain refinement strategies and the domain characteristics. Such evaluations show why a strategy performs better in some problems and worse in other problems.

In my evaluations of refinement strategies, I often constructed artificial domains where I could vary some factors in the problem that affected the performance of the tested refinement strategies. In addition, I tested the same strategies on existing domain problems such as the UM Translog problems. Often, the results from experiments on the artificial domain helped us explain the performances of the same strategies on these other test domains. These performance evaluations also led us to realize where the strategies could be improved upon. In addition, we performed statistical tests to see whether the differences in the performances of two strategies were significant or not, when such tests were applicable.

1.4 Contributions

Through the analyses of refinement strategies for HTN planning, I have produced the following results:

- **Least-commitment versus FAF.** Since the least commitment strategy, the most popular strategy in refinement planning, delays certain commitments, it may make premature decisions on other elements of a plan. Thus, the least commitment strategy may not perform well on all types of planning problems. However, the problem can be resolved in many cases by applying a different heuristic called the fewest alternative first (FAF) heuristic. Empirical tests show that there is a domain where one type of least commitment strategy performs well and another domain where another type of least commitment strategy performs well, but neither of the strategies performs well on both domains. However, a strategy called DVCS which chooses between these least commitment strategies based on the FAF heuristic performs as well as the least commitment strategy which performs best for each domain. Furthermore, there is a domain where DVCS outperforms both of the least commitment strategies.

- **Refinement as AND/OR tree search.** Different refinement strategies can be viewed as different ways to transform an AND/OR tree into an OR-tree. The efficiency of a refinement strategy can then be evaluated by taking the size of the OR-tree the strategy generates. The difference between the best and worst serializations can be as big as an exponential to the height of the AND/OR tree. In an empirical analysis, FAF generated the smallest OR-tree possible by this evaluation method.

- **The ExCon heuristic.** Interactions between tasks in HTN planning are harder to handle than in action-based planning because an HTN planner may not know the conditions and the effects a task has until it is decomposed into a more detailed plan. On many planning problem domains, this causes a major inefficiency. I found that most of these inefficiencies can be resolved by handling a certain type of condition in plans. Also, the conditions of this type, called external conditions, can be automatically detected by the planner. Based on this study, I created a new refinement heuristic called ExCon that improves the efficiency of planning by detecting and handling possibly problematic interactions between tasks. In a comparison of this heuristic against the FAF heuristic, ExCon performed increasingly better than FAF on problems where there are many task interactions.

- **Analysis of the effects of ordering constraints.** The ordering constraints in HTN planning domains can make an impact on the planning efficiency. In particular, a planner can use the left-to-right (LtoR) heuristic, which plans in a way similar to a forward planner, to plan efficiently on problems where there are many ordering constraints. Also, empirical tests show that many ordering constraints are necessary for LtoR to perform well. Furthermore, LtoR can be combined with ExCon to take advantages of both heuristics. The resulting strategy performed well on most of the test domains.

1.5 Organization

Chapter 2 provides the background for my research. It describes many refinement strategies for action-based planning that have been presented over the years. The chapter presents basic refinement planning systems and summarizes the refinement strategies for action-based planning. It also introduces the HTN planning methodology.

Next, I present three different heuristics used for refinement strategies. Chapter 3 describes the "Fewest Alternative First"(FAF) heuristic and shows the empirical results of FAF used to select between variable commitment and task instantiation commitment. The chapter explains that the choices that FAF makes can be thought of as serializing AND/OR graphs and analyzes the strategy from that perspective.

Chapter 4 defines external conditions, a type of condition in HTN planning which can be used to detect task interactions, and describes the "External Condition"(ExCon) heuristic, which tries to deal with task interactions efficiently by carefully selecting which task to decompose next. The performance of ExCon is compared with the performance of the FAF heuristic on various problems.

Chapter 5 describes the third heuristic, the "Left-to-Right"(LtoR) heuristic. This heuristic takes step ordering information into account when choosing which task to decompose next. The heuristic works particularly well on problems where orderings between many tasks are specified. This is supported by empirical tests comparing LtoR, FAF and ExCon.

Finally, Chapter 6 summarizes the results of this dissertation.

Chapter 2

Refinement Planning

Many general-domain planning systems use refinement search, where a planner takes a goal as a skeletal plan and gradually refines it (i.e. adds details) until a detailed solution plan is found. Although how each planner represents and refines partial plans varies from one to another, many refinement planners, be they action-based or HTN, share basic structures. This chapter provides background on refinement planning, various refinement strategies, and HTN planning.

2.1 Refinement Search

The concept of refinement search can be viewed as partitioning the search space of possible solutions into smaller spaces until one containing a valid solution is found. The refinement search process can be represented by a refinement search tree where the root represents the whole search space and each child node in the tree represents the different area resulting from a partition of the area represented by the parent node. When a search area that contains only inconsistent possible solutions is found, it can be pruned from the search.

For example, consider a partially developed plan P in Figure 2.1. The search

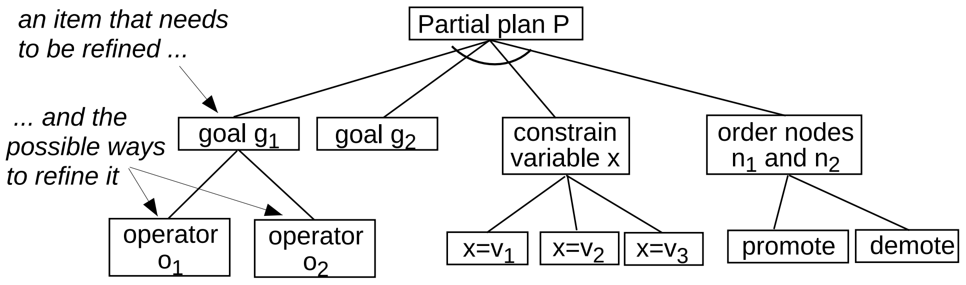


Figure 2.1: A sample refinement operations applicable to the partial plan P.

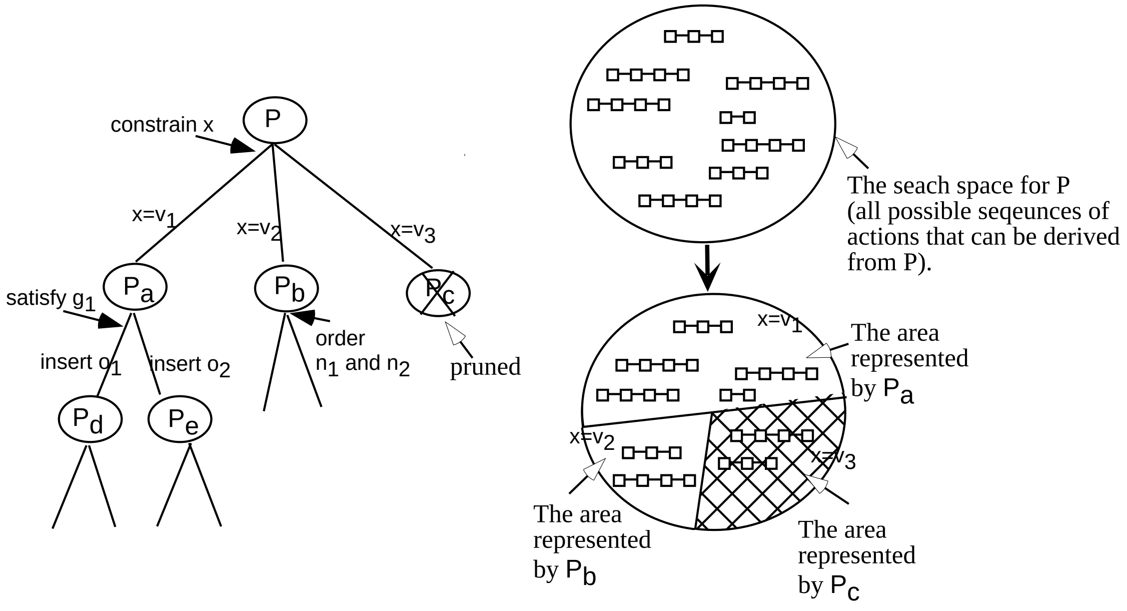

constrain x
P
x=v1
x=v2
x=v3
Pa
Pb
Pc
satisfy g1
order n1 and n2
pruned
insert o1
insert o2
Pd
Pe
The seach space for P (all possible seqeunces of actions that can be derived from P).
x=v1
The area represented by Pa
x=v2
x=v3
The area represented by Pb
The area represented by Pc
(a) The top of a refinement search tree
(b) The search space of possible solutions

Figure 2.2: An illustration of a refinement search to construct a plan from the partially developed plan P.

space of possible solutions for P is a set of all possible sequences of actions in the problem domain that can be derived from P. In order to develop P into a complete plan, a planner may refine P by inserting an action to satisfy the goal g_1 or the goal g_2, constrain the variable x, or put an ordering between the two nodes n_1 and n_2. Suppose the planner chooses to constrain the variable x first, the refinement will result in three different partial plans, each with a different value assigned to x (P_a, P_b, and P_c in the refinement search tree in Figure 2.2(a)). However, upon examining the plan P_c, the planner may find out that no consistent complete plan can derived from it. In such a case, P_c can be pruned from the search since no solution plan can be found by working on P_c. Figure 2.2(b) illustrates the corresponding partitioning of the search space for P. Now, suppose the planner next chooses to insert an action to satisfy the goal g_1. This refinement will result in two partial plans (P_d and P_e in Figure 2.2(a)).

Since searching an area that does not contain a valid solution is futile, the ability to identify whole areas that can be pruned from the search is very important for the efficiency of the refinement search. A refinement search algorithm often pre-prunes a node from the search tree. It does not generate a child node if it is clear that the node is inconsistent with the problem. Typically, the number of search nodes generated in the refinement search tree corresponds to the efficiency of the refinement algorithm. In addition, the efficiency of a refinement search algorithm is affected by the order the refinements are done. For example, if the first refinement operation for P was inserting an action to satisfy g_1, then the planner might later have to prune partial plans with $\mathsf{x} = \mathsf{v}_3$ twice at a later time, making the search less efficient. The techniques used to choose the ordering of the refinements that will improve the efficiency of planning are described in later chapters.

The basic algorithm of refinement search is as follows: node-set is a set of search nodes which need further refinements. In other words, the node-set represents different search areas in the search space that have not been pruned from the search. Initially, the node-set contains only a node with no refinements. The algorithm then repeatedly does the following: If the node-set is empty, the search is terminated with no solution because no solution exists for the given problem. If the node-set is not empty, one node n is picked nondeterministically and removed from the node-set. If n is a fully refined solution, the search is terminated with a solution. If not, n is refined with some refinement operation and the resulting nodes are put back in the node-set.

No pruning operations are explicitly done in the algorithm. Instead, inconsistent nodes are pre-pruned as part of the refinement operations. Since there are usually multiple refinement operations applicable to a node, a strategy, called a *refinement strategy*, is used to determine what refinement operation should be applied. As in other search algorithms, a refinement search algorithm is sound and complete if it returns a correct solution if and only if it exists.

Refinement planning uses refinement search to construct a solution plan. In refinement planning, a search node is a partially developed plan which typically consists of a set of action steps, orderings between the steps, a set of possible values for each variable, and other auxiliary constraints. A *completion* of a partial plan is obtained by instantiating every variable to one of its possible values and arranging the step orderings into a linear sequence in a way that satisfies all the associated constraints. A partial plan is *inconsistent* if it has no consistent completion. A partial plan is a *solution plan* if every completion solves the given problem.

2.2 Refinement Strategies for Action-based Planning

Many refinement planning systems, regardless of being action-based or HTN, use *least commitment strategy* techniques, which make certain decisions only when the planner thinks they are necessary. This section presents how action-based refinement planners generate plans, describes various least commitment strategies and discusses their advantages and disadvantages, and gives an outline of two other types of strategies for action-based planning, called Graphplan and SATPLAN, that are currently getting a lot of attention.

2.2.1 Refinements in action-based planning

Most planning formalisms, whether action-based or HTN, are based on the following assumptions:

- The planner has complete knowledge of the initial state of the world. Generally, an initial state is specified with a set of positive ground atoms, each stating a condition that is true in the world initially. Any condition that is not in the state is presumed to be false.
- The only thing that changes the state of the world is the execution of actions in the plan generated by the planner. If some condition is true in a state and some action is applied that does not affect the condition, then the condition will remain true in the resulting state.
- The planner knows all the consequences that each action execution makes.

- There is no gradual or delayed effect: all the effects of an action take place the instant the action is executed.[1]

In action-based planning, a planning problem is specified by a tuple $< I,\ g,\ D >$ where I is a initial state of the world, g is a goal and D is a set of domain operators. A *world state* is a set of positive ground atoms which represent descriptions of objects such as "BlockA is a block" or relationships between objects such as "BlockA is on top of BlockB". Any condition which does not appear in the state is assumed *False*. A *goal* is a partial description of the world state that a solution plan should achieve. In other words, all the conditions in the goal should be true in the state that can be achieved by executing the actions of a solution plan for the problem. A *domain operator* in action-based planning is the specification of an action in the domain. An operator is specified by preconditions which must be true in the world state in order to execute the action, and effects which represent the changes in the world state caused by the execution of the action. A *solution plan* for a problem is a sequence of instantiated domain operators which, when executed sequentially from the initial state, reach a final state where all goal conditions are necessarily true. A refinement planner can plan forward by gradually adding actions in the plan starting from the initial state, or plan backwards by gradually inserting actions from the end of the plan which satisfy goal conditions.

Example: action-based planning domain

Consider the following problem $P = < I,\ g,\ D >$:

I :Initial state = {(*on-table* a)(*clear* a)(*on-table* b)(*on* c b)(*clear* c)}

[1]There are some planning formalisms that allow conditional or probabilistic effects [35, 39] while some others allow gradual or delayed effects [51].

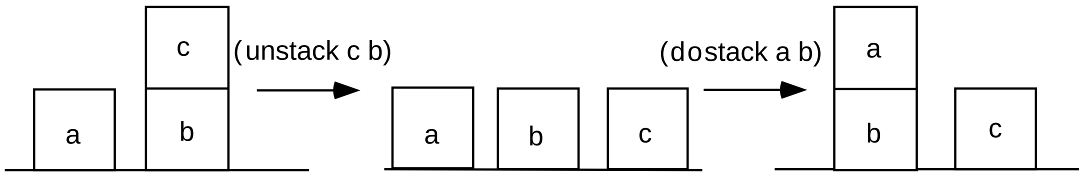


Figure 2.3: A plan for Blocks World problem.

g :Goal = {(*on* a b)}

D :Domain operators = {

unstack (?x ?y)

precondition: ((*clear* ?x)(*on* ?x ?y))

effect: ((∼*on* ?x ?y)(*on-table* ?x)(*clear* ?y))

dostack (?x ?y)

precondition: ((*clear* ?x)(*on-table* ?x)(*clear* ?y))

effect: ((∼*on-table* ?x)(*on* ?x ?y)(∼*clear* ?y))

restack (?x ?y ?z)

precondition: ((*clear* ?x)(*on* ?x ?y)(*clear* ?z))

effect: ((∼*on* ?x ?y)(*on* ?x ?z)(∼*clear* ?z)(*clear* ?y))

}

There are three blocks, a, b and c. Initially, blocks a and b are on the table and block c is on top of block b. The goal of this problem is to stack block a on top of block b. There are three domain operators available. unstack puts a block which is on top of another block down on the table. dostack stacks a block from the table on top of another block. restack moves a block from on top of a block to the top

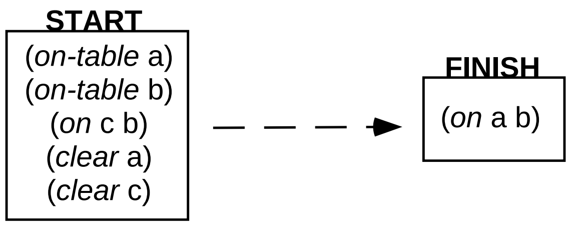


Figure 2.4: An initial partial plan for the Blocks World sample problem.

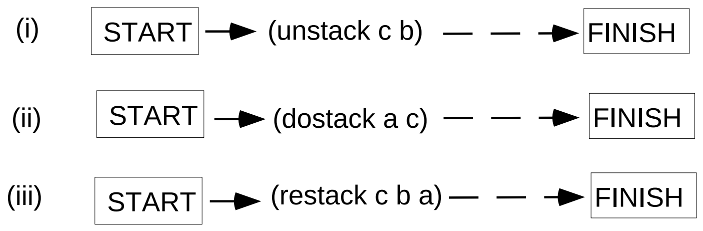


Figure 2.5: Three alternative partial plans that can be obtained by asserting an action applicable to the initial state.

of another block. One solution plan for the above problem is the sequence (unstack c b);(dostack a b) which is illustrated in Figure 2.3. First, (unstack c b) clears the top of the block b, then (dostack a b) makes (*on* a b) true in the final state.

A refinement planner usually starts with an initial partial plan consisting of two steps, START and FINISH as shown in Figure 2.4. START is a dummy action with no preconditions and the effects establish all the conditions of the initial state. FINISH is a dummy action with no effects and the goal conditions are its preconditions. Every action the planner asserts is placed between START and FINISH. The goal of the planner is to construct a sequence of actions where any precondition of an action is made true by an effect of a preceding action without any other action in between negating the effect.

Typically, a planner has two ways to assert an action in an partial plan. One way is to 'plan forwards' by asserting an action that can be applied right after

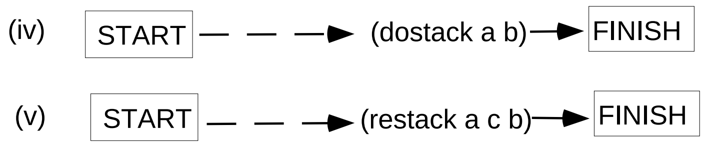


Figure 2.6: Two alternative partial plans that can be obtained by asserting an action that establishes the goal condition.

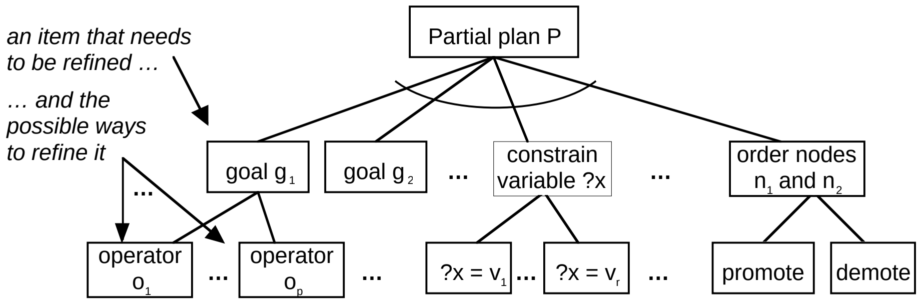


Figure 2.7: Various options for refinement operations.

the START action or the subsequent actions. Another is to 'plan backwards' by asserting an action that can satisfy one of the preconditions of existing actions. For example, a planner which plans forward would assert (unstack c b), (dostack a c) or (restack c b a) to the plan after START (Figure 2.5), while a planner which plans backward would assert (dostack a b) or (restack a ?y b) to the plan before FINISH (Figure 2.6). Typically, a planner only plans forward [7], or only plans backward [31, 8], although some planners plan bi-directionally [16].

In addition to asserting actions, a planner needs to check the consistency of the partial plan, bind variables, and detect and resolve conflicts. Thus, a planner has various options for what to do during the planning process as shown by Figure 2.7. As in general refinement search, a refinement strategy determines how and in what order these refinements are done.

2.2.2 Least Commitment Strategies

As AI planning evolved, various types of refinement strategies were introduced and improved. The earliest planning systems like STRIPS [15] use the linear planning method where partial plans are linear sequences of steps. Also, the variables in an operator are instantiated into constants as soon as the operator is added as a step in the plan.

The notion of 'least commitments' was first introduced by Sacerdoti [41]. In his planning system called NOAH, he used a partially-ordered graph to represent the step orderings in a partial plan. This provided a way to avoid premature commitments to a particular step ordering when achieving subgoals. Compared with the previous method of using only linear sequences of steps in partial plans, using partially ordered steps has been shown to reduce the search space and thus improves the planning efficiency [33]. Over the years, the "avoid premature commitments" idea has been extended in various ways to many types of commitments, including commitments to variable bindings and commitments to subgoal reductions. Since many people have defined how and when a type of commitment is 'premature' in so many different ways, a 'least commitment strategy' should be regarded not as one unique strategy, but as a strategy that was created based on the "avoid premature commitments" concept. Thus, a 'least commitment strategy' generally refers to any strategy that tries to avoid unnecessary branching in the search by either postponing certain refinements or making only necessary changes to partial plans by adapting a flexible representation of a partial plan.[2] We will revisit the issue of

[2]In contrast to least commitment, a refinement strategy which tries to make decisions as soon as possible in the plan is called an 'eager commitment' strategy or a 'maximum commitment' strategy.

least commitment strategies in Chapter 3 and carefully examine their efficiency.

Least commitment strategies are the most popular strategies for refinement planning. Below is a list of the major least commitment techniques. Some of the techniques were introduced in the HTN planning framework, although they were later used in action-based planning. Similarly, many of the techniques that were created in action-based planning have been applied in HTN planning.

- **Step orderings:** Instead of always representing a partially developed plan by a linear sequence of steps, Sacerdoti [41] used a partially ordered graph to avoid unnecessary commitments to step orderings. In order to keep a plan free of conflicts between steps that are not ordered with each other, Chapman [8] presented the Modal Truth Criterion (MTC) for evaluating conditions in his TWEAK planner. MTC determines if a condition p is *necessarily true* to denote p is satisfied, *possibly true* to denote p can be possibly satisfied later, or *false* to denote p cannot be satisfied. TWEAK terminates with a solution plan when all the conditions in the plan are necessarily true. An alternative to MTC is to use 'causal links' which were introduced by Tate [47]. A causal link records a causality between two steps and can be denoted by a tuple $< p,\ s_e,\ s_c >$ where s_e is the establishing action step for a condition p which is a precondition of the consuming step s_c. A causal-link based planner like SNLP [31] creates a causal link each time a subgoal (i.e. an unsatisfied precondition of a step in the plan) is satisfied by another step. Unlike MTC, which repairs threatening situations by inserting another establishing step between the threatening step and the consuming step, the causal links in a plan cannot be violated; when a causal link is found to be violated in the plan, then the plan is considered inconsistent and pruned from the search.

- **Variable binding:** In traditional planning system like STRIPS [15] and Nonlin [47], the variables in an operator are instantiated into constant values immediately after the operator is introduced in a partial plan. Yang and Chan [56] pointed out that this approach to variable bindings has several drawbacks. First of all, this approach creates 100 branches in the search tree if a variable has 100 possible values. The large branching factor causes an enormous search space. Second, if the planner employs chronological backtracking, the search might keep failing for the same variable binding decision. The suggestion Yang and Chan presented is to maintain the sets of possible values for the variables instead of binding them to constant values. They presented a new planning algorithm FSNLP, an extension of SNLP, which delays variable bindings until they become absolutely necessary. Instead of instantiating variables into constant values as soon as a step is inserted to a partial plan, FSNLP keeps the sets of possible values for variables. In order to detect partial plans where there is no consistent instantiation, FSNLP occasionally calls a CSP(Constraint Satisfaction Problem) algorithm to check if there exists a consistent assignment of variables. Their experiments on the Smith and Peot domain [42] with varying number of objects showed FSNLP performed better than SNLP.

- **Subgoal reduction:** Friedman and Weld [17] developed a least commitment technique for subgoal reduction. Unlike other planners such as SNLP where the search branches out for the number of operators applicable to the subgoal, their planner FABIAN uses an abstract operator which represents applicable operators when establishing a subgoal and the search branches out only when necessary. The abstract operator of a predicate represents a disjunction of possible new establishing operators. Abstract operators are constructed in polynomial time preprocessing for each predicate. Friedman and Weld showed that this approach can lead to exponential savings while it never explores more than a roughly logarithmic larger space than SNLP.

- **Constraint posting:** Sometimes, fully satisfying constraints such as a variable inequality constraint (i.e. $?x \neq ?y$) results in many search branches. In the MOLGEN [45] planner, Stefik introduced the constraint posting approach. Instead of the planner fully satisfying constraints immediately by binding variables, the constraints are *posted* and updated in the plan so that the planner can defer variable binding decisions until the variables are constrained more. Another example of constraint posting is the technique of keeping causal links in a causal-link planner as described above.

- **Conflict resolution:** In the SNLP planner, all threats are resolved at every refinement cycle before satisfying the next open condition in the partial plan. There are three ways to resolve a threat to a causal link in SNLP; *demotion*, *promotion* and *separation*. Demotion moves the threatening step so it is before the step that produces the effect for the causal link. Promotion moves the threatening step so it occurs after the step whose precondition the causal link protects. Separation binds variables so that the effects of the threatening step and the corresponding condition of the causal link will not unify. This method usually creates three or more partial plans in the search space. Peot and Smith [37] observed that since other necessary planning operations often make the threatening situations go away, it is often a good idea to delay threat removals. They developed four threat removal strategies: 'DSep' defers a threat removal until the threat cannot be removed by separation; 'DUnf' defers a threat removal until there is only one threat resolution option remaining; 'DRes' does no threat resolution, it simply discards a partial plan when an unresolvable threat is found; and 'DEnd' resolves a threat only when all the open conditions have been satisfied. Their empirical results on several domains showed that DSep and DUnf did better than DRes, DEnd, or the default strategy of SNLP.

Effectiveness of least commitment

There are several comparisons of least commitment against maximum commitment. Minton, et al. [33] discussed ordering commitments from the perspective of partial-order planning vs. total-order planning. They compared a total-order planning algorithm TO against a partial-order planning algorithm UA. The two algorithms are constructed such that each node of the search tree TO generates represents a linearization of a node at the same depth of the search tree UA generates. Thus with breadth-first search, the two algorithms can be synchronized (i.e. they search through the same parts of the plan space). It was shown that the size of a search tree for UA is less than the one for TO regardless of the domain. Therefore, the search space of UA is proven to be smaller than the search space of TO with breadth-first search. The same analysis cannot be applied under depth-first search, although experiments in the Blocks World domain showed UA performs significantly better than TO. Minton, et al. also argued there is a correspondence between search strategies and the performance of partial-order planning. They showed that UA can take advantage of certain types of heuristics more effectively than TO. Despite the analytical results of UA and TO, Minton, et al. pointed out a partial-order planner has a potentially much larger search space than a total-order planner since there are many more partial orderings over a set of steps than there are total orderings; thus, the performance of a partial-order planner relies on more intelligent ordering choices. Barrett and Weld [4] also compared partial-order vs. total-order planning. They used a causal-link total-order planning algorithm TOCL and a causal-link partial-order planning algorithm POCL. In contrast to Minton, et al's work on domain independent analysis, Barrett and Weld focused on the domain structures. More specifically, they argued the serializability of the domain is the key factor in the performance difference between TOCL and POCL.

In many discussions of refinement strategies, the least commitment strategy is generally favored over maximum commitment. Veloso and Stone [50], however, showed by experimental results that delayed-decision commitments do not always mean better performance in comparison with other commitment strategies. They presented FLECS, an extension of the Prodigy planner, as a framework for their analysis on ordering commitment strategy. Prodigy keeps a totally-ordered head plan and a partially-ordered tail plan as an internal plan structure. Subgoaling does backward chaining starting from the goal, using means-end analysis. The operator introduced in the plan is then applied to the totally-ordered head plan if the all operator's preconditions are true in the end state of the head plan. In FLECS, commitment strategies are defined by a 2-value toggle which decides between subgoaling or applying an operator to the totally-ordered plan. Veloso and Stone ran the experiments in three domains. In the first domain, eager subgoaling gave better CPU time than eager applying. In the second domain, eager applying gave better CPU time than eager subgoaling. And in the third domain, a strategy which makes a choice according to the current state of the plan worked significantly better than either eager subgoaling or eager applying. The results clearly showed that the performances of commitment strategies are domain dependent.

Kambhampati, et al. [23] provided a unified framework for partial-order planners as a convenient base for analyzing different design choices in partial-order planning algorithms. The algorithm Refine-Plan-PO is a planning framework which can be used to implement planning algorithms by providing the methods for termination, goal selection, precondition establishment, bookkeeping, consistency check and conflict resolution. Using the Refine-Plan-PO framework, Kambhampati, et al. compared major existing domain independent partial-order planners including UA, SNLP, Tweak [8], UCPOP [36] and several other hybrid planning algorithms. They argued that the relative performances of an eager commitment strategy and a least commitment strategy depend on the presence of *high-frequency conditions*—conditions that have many establishers (i.e. steps that can establish the condition) and threats (i.e. steps that deny the condition). They hypothesized that if none of the conditions in the domain are high-frequency conditions, least commitments do better than eager commitments while if most of the conditions in the domain are high-frequency conditions, eager commitments do better than least commitments. These hypotheses were supported by their experimental results.

2.2.3 Other strategies

Recently, graph-based planning and SAT-based planning for action-based operators have been getting much attention because their planning speed was shown to be much faster than classical action-based planners.

In Graphplan [7], developed by Blum and Furst, the planner constructs a planning graph which represents every possible state that can be achieved from the initial state. When the all goal conditions are found in a state, the planner retraces the graph to find a solution. In SATPLAN [25], an action-based planning problem is first translated into an equivalent propositional logic problem, and then solved by a fast general propositional satisfiability solver.

Although there have been several efforts to combine Graphplan and HTN planning [28] or SATPLAN and HTN planning [30], it has not yet been shown how effective they are compared with traditional HTN planning. Another concern is that many planning applications need to interleave planning and execution. One advantage of an HTN planning framework is that it allows the planner to work on a portion of a plan more easily than an action-based framework would. However, while most refinement strategies, including the ones presented in this paper, can be applied to the portion of a plan, it is not clear if the Graphplan or SATPLAN technique can be applied likewise. Thus, using the Graphplan or SATPLAN technique in HTN planning may probably be effective in some applications but not feasible in others.

Recently, Kambhampati [22] presented the view that both Graphplan and SATPLAN are a special type of refinement planning where they use refinements to prune the search space but not to partition it. Overall, whether or not these new techniques are better than traditional refinement planning techniques is still an open question.

2.3 HTN Planning

HTN planning started with Sacerdoti's use of 'procedural nets' in his planning system called NOAH [41]. A procedural net is a network of tasks representing a partial plan where each node represents a particular action at some level of detail.

NOAH plans by repeatedly (1) decomposing abstract tasks in a plan into more detailed tasks, and (2) detecting and removing conflicts in the resulting partial plans. The latter operation is done by what are called ‘critics’. The term ‘task network’, introduced by McDermott [32], refers to a set of tasks linked together by plan properties such as temporal ordering and state conditions. Thus, hierarchical task network (HTN) planning refers to a planning methodology where partial plans are represented by task networks and actions in a plan are acquired by decomposing abstract tasks.

In action-based planning, domain operators are specified by their preconditions and effects. An action-based planner plans by adding operators into a plan so that the goal conditions are achieved and the plan is conflict-free. An HTN planner proceeds by decomposing abstract tasks into more detailed tasks by applying decomposition methods. Even though HTN planning is used more in practical applications, most studies in AI planning have been done in the action-based planning framework. This was in part due to the lack of a solid framework for HTN planning until recently; for example, there was no sound and complete HTN algorithm until the work by Erol, et al. [12]. Their UMCP algorithm is provably sound, complete and systematic. We have used its implementation, the UMCP planner, as a testbed to test various refinement strategies.

In this section we present the basic mechanism of HTN planning and the basic planning methods in UMCP. We also describe other HTN planning frameworks and compare the advantages and disadvantages of action-based planning and HTN planning.

2.3.1 Illustration of HTN planning

Take a problem we call "a college student's dinner problem". The goal of the problem here is to eat pizza. Assume this takes place when one is in his apartment and he does not have a box of frozen pizza in the freezer. There are several alternative ways he considers; (a) eat in a pizza restaurant, (b) order by phone for a delivery, or (c) buy frozen pizza and heat it. Upon deciding which one of the alternative ways to use, more detailed plans can be made; for example, option (b) above can be accomplished by getting the phone number of a pizza place, calling and ordering a pizza, paying for the pizza when delivered, and eating the pizza. Moreover, each of these actions, such as getting the phone number, needs to be expanded into more detailed actions, until the entire plan consists of only executable actions. An illustration of this planning process is shown in Figure 2.8. Also, planning has to take into account various constraints associated with the actions. For example, if he wants to watch TV that night, then the option (a) of eating out may not be applicable. Also, if one's microwave or oven is out of order, then the option (c) of buying frozen pizza and heating it would not work.

Many plans can be created in a hierarchical manner similar to the above planning process. An HTN planner generates plans the same way but more methodically. As such, HTN planning is suitable for many real-world planning domains since many planning decisions made in the real world are done in a hierarchical manner. Also, it is easy to transform a domain specification for action-based planning into an equivalent HTN domain specification [12].

For an HTN planner, there are two types of actions. One is a primitive action (also called a primitive task) which a robot or a human can execute without further planning. The other is a non-primitive action (also called a non-primitive

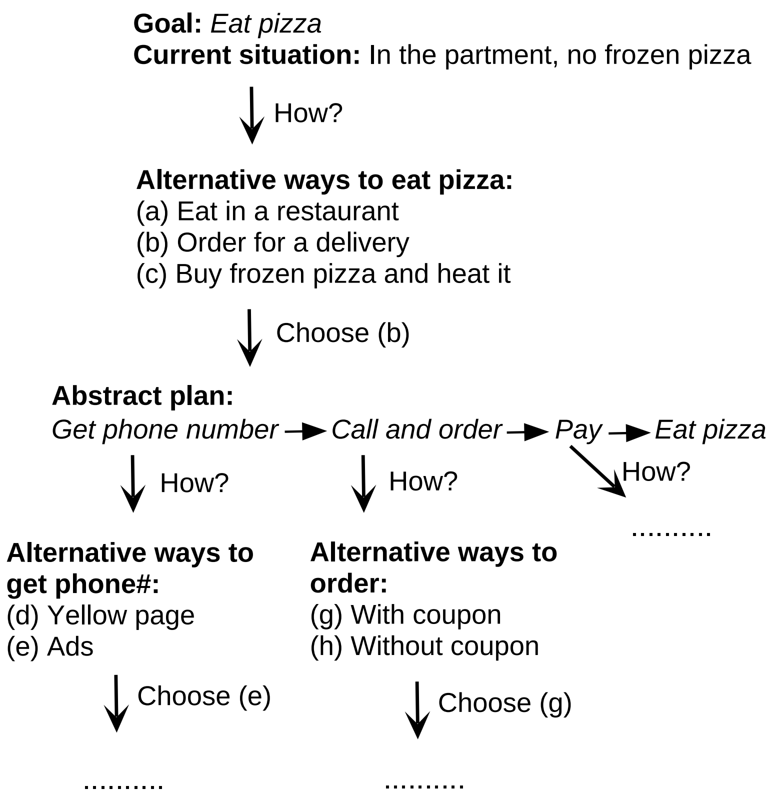
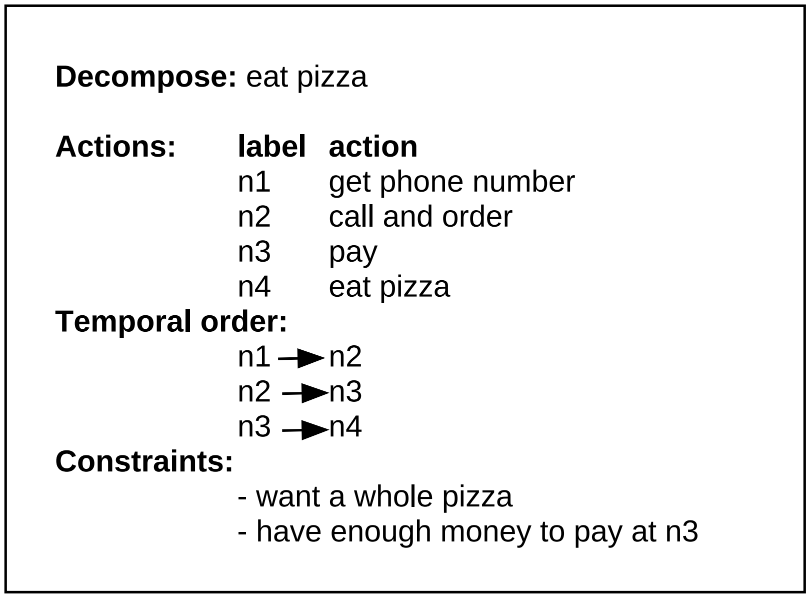

Goal: *Eat pizza*
Current situation: In the partment, no frozen pizza
How?
Alternative ways to eat pizza:
(a) Eat in a restaurant
(b) Order for a delivery
(c) Buy frozen pizza and heat it
Choose (b)
Abstract plan:
Get phone number → *Call and order* → *Pay* → *Eat pizza*
How?
How?
How?
....
Alternative ways to get phone#:
(d) Yellow page
(e) Ads
Alternative ways to order:
(g) With coupon
(h) Without coupon
Choose (e)
Choose (g)
....
....

Figure 2.8: A college student's dinner problem



Decompose: eat pizza

Actions:

label	**action**
n1	get phone number
n2	call and order
n3	pay
n4	eat pizza

Temporal order:

n1 $\rightarrow$ n2
n2 $\rightarrow$ n3
n3 $\rightarrow$ n4

Constraints:

- want a whole pizza
- have enough money to pay at n3

Figure 2.9: An informal description of a decomposition method for the action 'eat pizza'

task or abstract action), which the planner needs to 'decompose' into more detailed actions. Decomposing a non-primitive action is accomplished by applying a *decomposition method* for the action. A decomposition method prescribes the actions that the non-primitive action is decomposed into, the orderings between these actions, and various other constraints that need to be satisfied by the planner.

For example, consider describing the actions for option (b) in a decomposition method. One method description might be something like Figure 2.9. There are four actions necessary to accomplish the action 'eat pizza'. Every action in a plan or a decomposition method has a unique label to distinguish between multiple instances of an action, and also to simplify the specifications. So, for example, the action 'get phone number' is labeled `n1` in this decomposition method. The temporal ordering of actions are also specified in the method using action labels.

In this method, expressing ‘get phone number’ should be done before ‘call and order’, which should be done before ‘pay’, which should be done before ‘eat pizza’. Also, there are two constraints: that the student wants a whole pizza (not a slice) and that there is enough money to pay for the pizza when paying.

2.3.2 Basic HTN planning mechanism

An HTN planner generates a plan by decomposing tasks into more detailed tasks, enforcing constraints and resolving conflicts until all the tasks in the plan are *primitive* (i.e. executable actions) and there are no conflicts between tasks. Figure 2.10 shows a basic HTN algorithm. The same assumptions as in action-based planning about the world state, as described in section 2.2.1, hold in HTN planning. In HTN planning, two types of goals can be expressed in a problem. One is a condition goal which must be made true at the end of the solution plan, and the other is an abstract action which must be accomplished by a solution plan. An HTN problem can contain one or more goals and also specific constraints associated with the goals.

In general there are two types of tasks in HTN planning. A *primitive task* is an executable action which does not need any decomposition. A solution plan should contain only primitive tasks. A *non-primitive task* is an abstract task which needs to be decomposed. Given a planning problem P, the planner repeatedly checks if P is a solution or if P is inconsistent (Step 2) and terminates in either case, picks a non-primitive task in P and a decomposition method for the task (Steps 3 and 4), applies the decomposition method to the task (Step 5), and handles task interactions (Steps 6 and 7). Normally, there is more than one way to decompose a task, and there is more than one way to handle task interactions in a plan, thus

```
Algorithm Refine-HTN()

  1. Input a planning problem P.
  2. If P contains only primitive tasks, then resolve the conflicts in P and
     return the result. If the conflicts cannot be resolved, return failure.
  3. Choose a non-primitive task t in P.
  4. Choose a decomposition for t. (backtracking point)
  5. Replace t with the decomposition.
  6. Use critics to find the interactions among the tasks in P, and suggest
     ways to handle them.
  7. Apply one of the ways suggested in step 6. (backtracking point)
  8. Go to step 2.
```

Figure 2.10: Basic HTN planning algorithm. [13]

making choice points in the search.

2.3.3 UMCP Formalism

The UMCP algorithm is based on a generalization of HTN planning [12]. This section presents the UMCP formalism [11] and shows the types of refinement operations used in HTN planning. This section is also intended to provide the terminology which is used in later chapters.

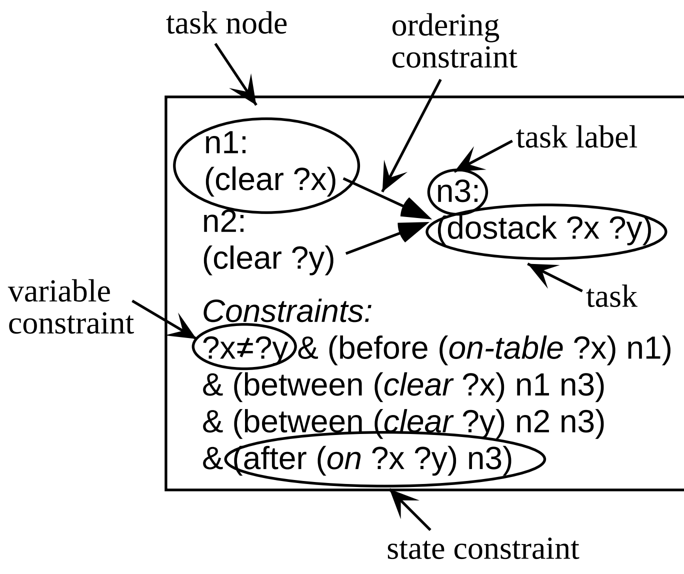


Figure 2.11: A method for (*on* ?x ?y) in Blocks World (in UMCP specification).

Task Networks

In UMCP, both partial plans and decomposition methods are represented in the form of task networks. A task network is a partially specified plan that contains partially ordered tasks and various constraints associated with them. Figure 2.11 shows a sample task network; a decomposition method to achieve the condition (*on* ?x ?y) in the Blocks World domain which was used as an example of action-based planning domain in Section 2.2.1. The decomposition uses the same three predicates *on*, *clear* and *on-table* to specify state conditions. So (*on* ?x ?y) refers to "block ?x is on block ?y", (*clear* ?x) refers to "there is no block on top of block ?x", and (*on-table* ?x) refers to "block ?x is placed on the table". In order to accommodate goal conditions and preconditions that are used in action-based planning, UMCP has a special type of non-primitive task called *predicate tasks*. A predicate task has the form of a positive or negative literal, and its main purpose is to satisfy the condition referred by the literal. For example, the above sample method is a method for the predicate task (on ?x ?y) that will make the condition (*on* ?x ?y) true.[3] Also, there are two predicate tasks (clear ?x) and (clear ?y) in the method to specify that the conditions (*clear* ?x) and (*clear* ?y) need to be made true before doing the task (dostack ?x ?y). A predicate task is similar to a supervised condition introduced by Tate [47] in 1977. In order to distinguish other non-primitive tasks from predicate tasks, non-primitive tasks that are not predicate tasks in UMCP are called *compound tasks*.

In the above decomposition method, each task is labeled ni for some i. In UMCP, every task in a partial plan or decomposition method has a unique label, in order to distinguish multiple instances of the same task. These labels are used in the constraint specification to specify various constraints associated with the related tasks.

There are four constraints specified in the method. The constraint ?x $\neq$?y specifies that the variables ?x and ?y cannot have the same value. The constraint (before (*on-table* ?x) n1) specifies that the block ?x must be on the table before doing the task referred to by n1 (i.e. (clear ?x)). The constraint (between (*clear* ?x) n1 n3) specifies that the block ?x must have no blocks on top anytime after doing the task referred to by n1 (i.e. (clear ?x)) until starting the task referred to by n3 (i.e. (dostack ?x ?y)). Similarly, the constraint (between (*clear* ?y) n2 n3) specifies that the condition (*clear* ?y) must persist between the time interval between task n2 and task n3. The fourth constraint (after (*on* ?x ?y) n3) states that after finishing the execution of the task referred by n3 (i.e. (dostack ?x ?y)), the condition (*on* ?x ?y) must be true.

Notice the similarity between predicate tasks and some of the constraints. In

[3]In order to distinguish tasks from state conditions, predicate names are written in italics. For example, (on ?x ?y) is a predicate task while (*on* ?x ?y) is a state condition.

the above method, the predicate task (clear ?x) specifies "the condition (*clear* ?x) needs to be made true before doing the task (dostack ?x ?y)". The constraint (before (*on-table* ?x) n1) specifies that "the block ?x must be on the table before doing the task referred to by n1 (i.e. (clear ?x))". Both conditions ((*clear* ?x) and (*on-table* ?x)) need to be satisfied in the specific state in order for the decomposition method to work. Why is one specified by a predicate task and another specified in a constraint? The difference between using a predicate task and using a constraint is how the condition should be made true. If a condition is specified by a predicate task and the condition is not true in the initial state or by the effects of other tasks, then the planner actively tries to make it true by inserting actions specified in the decomposition method for the predicate task. On the other hand, a condition specified in a constraint has to be true without the planner inserting actions into the plan. The planner can try to satisfy the condition in a constraint by ordering tasks in the plan or binding variables, but the planner cannot assert new actions to satisfy it. Having these two types of condition specification is useful for many reasons. For example, in the above Blocks World example, the planner does not use the method if block ?x is not placed on the table. Because, if it is on the table, then the planner should use a method that specifies a restack action instead of a dostack action. On the other hand, specifying the condition (*clear* ?x) by a predicate task is convenient because the planner can then plan for the necessary actions to prepare that the task (dostack ?x ?y) would work if the condition is not true.

More formal definitions of the elements of a task network are as follows:

- **Tasks:** A task $t(x_1 \cdots x_k)$ is either primitive, predicate, or compound. A *primitive* task is an atomic action that can take place in the world. When executed, it may have effects $\{l_1, l_2, \cdots, l_n\}$ where each l_i is a literal whose arguments are either constants or variables x_j. A *predicate* task has the form $p(x_1 \cdots x_k)$ where p is a positive or negative predicate symbol. Its main purpose is to achieve the condition $p(x_1 \cdots x_k)$ and it has one or more decomposition methods which specify how to achieve the condition. If the condition is already achieved in the plan without executing specific actions to make it true, then the task is *phantomized*, i.e. the task is replaced by an empty action. A *compound* task is an abstract action. Like a predicate task, it can be decomposed into more detailed tasks by applying a decomposition method.

- **Constraints:** There are four types of atomic constraints: *Variable binding constraints* are of the form $(v1 = c)$, $(v1 = v2)$, $(v1 \neq c)$ or $(v1 \neq v2)$ where $v1$, $v2$ are variables and c is a constant. They represent constraints on variable (non-)codesignations; *Ordering constraints* are of the form $(ord\ n\ n')$, where n and n' are either a node label, which represents a specific task in the task network, or a node expression. The constraint specifies that the task labeled with n must finish before the beginning of the task labeled with n'. A node expression is either of the form (first $n_i\ n_j\ \cdots$) or (last $n_i\ n_j\ \cdots$), referring to the node whose task starts first or end lasts among the tasks in the list; *State constraints* are of the form (before $l\ n$), (after $l\ n$) or (between $l\ n\ n'$) where n and n' are node labels and and l is a literal. (before $l\ n$) or (after $l\ n$) is true if l is true immediately before n, or immediately after n. (between $l\ n\ n'$) is true if l is true from the end of n through the beginning of n'; *Initial state constraints* are of the form (initially l) where l is a literal which has to be true in the initial state. A *constraint formula* is a Boolean formula constructed from atomic constraints described above and conjunctive, disjunctive and negative operators.

- **Task Networks:** A task network has the form $< T,\ \phi >$, where T is a set of task nodes and ϕ is a constraint formula. A task node is a tuple $< n_i,\ t_i >$ where n_i is the node label for the task t_i. If a task network contains only primitive tasks, then it is called *primitive task network*.

- **Decomposition Methods:** A decomposition method for a non-primitive task t is a task network $m = < T_i,\ \phi_i >$. Decomposing an instance of t in a partial plan (i.e. task network) $P = < T,\ \phi >$ results in a new partial plan $P_{new} = < (T - \{< n_t,\ t >\}) \cup T_{inew},\ \phi \wedge \phi_{inew} >$ where n_t is the label for the task t in P, T_{inew} and ϕ_{inew} are T_i and ϕ_i, respectively, with necessary variable bindings.

- **Initial states:** An initial state I is a set of ground positive atoms which represent what conditions are true initially. The planner assumes any atom which is not in I to be false.

Refinement Search in UMCP

Figure 2.12 shows the algorithm for high-level refinement search in the UMCP planner. A problem is a tuple $< g,\ I,\ D >$ where I is an initial state, g is a task network representing the goal, and D is the problem domain. A world state in HTN planning is the same as one in action-based planning; it is a set of ground atoms in which only the execution of an action can make changes. A domain consists of a set of primitive task specifications, and a set of decomposition methods for non-primitive tasks. In the UMCP planner, a task-network contains an auxiliary data structure which stores step orderings, sets of possible values for variables, and a list of delayed constraints called the Promissory List, in addition to a set of task nodes and a constraint formula.

UMCP gradually refines task-networks representing partial plans, by decomposing non-primitive tasks, enforcing constraints, and checking consistency, until a task network is found where every task is primitive and all the constraints are satisfied in the data structure of the task network. The high-level refinement search is implemented with the A* algorithm where un-explored search nodes are kept in OpenList. UMCP repeatedly removes a task network from OpenList, and applies a refinement strategy R which may do any combination of planning operations such as task decomposition and consistency checking. The task networks returned by R are put back into OpenList. UMCP allows the user to choose depth-first search, breadth-first search, or best-first search by changing the method used to pick a node in Step 4.

Since the refinement strategy R can make any changes to the task network, it needs to satisfy the following requirements in order for the UMCP algorithm to be sound and complete.

- Soundness: Any solution to any task network in $R(tn)$ must also be a solution for tn. UMCP is sound if the refinement strategy that is used satisfies this property because commitments in task networks grow monotonically, and constraints in the Promissory List are removed only as they become necessarily true.

- Completeness: Any solution for tn must also be a solution for some task network in $R(tn)$. UMCP is complete if the refinement strategy that is used satisfies this property. Any time a constraint is selected in the constraint selection phase, its negation is also selected (unless it contradicts the

```
Algorithm UMCP()

  1. Input a planning problem P = $< g, I, D >$.
  2. Initialize OpenList to contain only $g$.
  3. If OpenList is empty, then halt and return "no solutions".
  4. Pick and remove a task network $tn$ from OpenList.
  5. If $tn$ is completely refined into a solution then halt and return $tn$.
  6. Pick a refinement strategy $R$ for $tn$.
  7. Apply $R$ to $tn$ and insert the resulting set of task networks into OpenList.
  8. Go to step 3.
```

Figure 2.12: High-level refinement search in UMCP algorithm. [13]

commitments or the constraint formula), and all possible ways of making a constraint true are tried in the constraint update phase.

The Default Refinement Strategy in UMCP

The default refinement strategy in UMCP employs several least commitment techniques. Each partial plan (task-network) keeps an auxiliary data structure in order to facilitate these techniques.

- **Possible values:** Similar to the method used in FSNLP [56], each variable in UMCP has a set of possible values associated with it in order to postpone premature variable instantiation. UMCP constrains the set when enforcing constraints. Suppose the variable v has the possible value set S. Then, enforcing the constraint $(v = c)$, where v is a variable and c is a constant, sets $S = \{c\}$ if c is in S, and returns *nil* otherwise to report the inconsistency. Enforcing $(v \neq c)$ removes c from S if c is in S, and make no changes otherwise. Enforcing $(v = v2)$, where $v2$ is also a variable, sets S to the intersection of S and the possible value set for $v2$ and also replaces the occurencies of $v2$ in the partial plan with v. It returns *nil* if the intersection is empty. Enforcing $(v \neq v2)$ makes no change if the intersection of S and the possible value set for $v2$ is empty. Otherwise, the constraint $(v \neq v2)$ is put into the Promissory List of the partial plan. Also, enforcing state constraints or initial state constraints can constrain the set S if the constraint contains v.

- **Partially ordered steps:** As described before, step orderings in a partial plan are represented in the form of a partially ordered graph. If enforcing a constraint makes a cycle in the graph, UMCP prunes the partial plan for inconsistency.

- **Constraint posting:** One of the major differences in constraint enforcement between an action-based planner and an HTN planner is that some constraints cannot be immediately enforced in a partial plan of an HTN planner if the partial plan contains non-primitive tasks. Some constraints can only be made true by further decomposing tasks in the partial plan.[4] In UMCP, some state constraints and ordering constraints may not be fully enforced immediately. UMCP keeps such constraints in the Promissory List of the partial plan. The Promissory List also contains non-codesignation

[4]Some HTN planners such as SIPE require users to formulate the domain specification such that all constraints can be established in a planning level.

variable constraints. Similar to the constraint posting technique used by Stefik [45], the constraints in the Promissory List are constantly updated and propagated.

Based on the current partial plan (i.e. task network), UMCP's default refinement strategy does one of the following actions: (1) Decompose a non-primitive task; (2) Enforce each of the newly inserted constraints; (3) Evaluate and simplify the constraint formula; or (4) Propagate previously postponed constraints. Figure 2.13 shows the algorithm that UMCP uses to decide which refinement to do next. It takes a partial plan as input and returns a set of partial plans as the result of the refinement performed.

If the constraint formula of the partial plan is *False*, then UMCP prunes the partial plan by returning an empty set in Step 1. When the constraint formula is *True*, UMCP decomposes a non-primitive task t in the partial plan at Step 2. Decomposing t involves, for each decomposition method M of t, (a) replacing t with the subtasks in M, and (b) replacing the current constraint formula C in the partial plan with the conjunction of C and the constraint formula in M. If t is a predicate task, t is also phantomized by creating a plan with the task t replaced with the do-nothing task and the constraint formula specifying that the predicate is accomplished at the beginning of the do-nothing task. Step 3 checks if the partial plan is a solution plan or not. If there are no non-primitive tasks in the partial plan and the constraint formula is *True*, then UMCP satisfies the remaining auxiliary constraints by instantiating variables and ordering steps, and returns the solution plans. If the constraint formula is neither *True* nor *False*, then UMCP enforces the constraints in the partial plan at Step 4. Enforcing constraints involves adding step orderings to the tasks and/or binding variables, according to the constraint

Algorithm refine(*PartialPlan*)

1. *(Pruning)* If the constraint formula of *PartialPlan* is *False* then prune this plan by returning empty set.

2. *(Task decomposition)* Otherwise, if the constraint formula is *True* and there are non-primitive tasks in *PartialPlan*, then decompose a task and return the resulting partial plans.

3. *(Solution check)* If the constraint formula is *True* and there are no non-primitive tasks in *PartialPlan*, then satisfy the auxiliary constraints and return the resulting plans as solutions.

4. *(Constraint enforcement)* If the constraint formula is neither *True* nor *False*, then satisfy constraints, simplify constraint formula, and propagate auxiliary constraints. Return the resulting partial plans.

Figure 2.13: The default refinement strategy in the UMCP planner.

types. If it requires further task decomposition to fully enforce some constraint, the constraint will be put into the Promissory List to be enforced later. If a constraint is not enforceable, then an empty plan is returned.

2.3.4 Other HTN formalisms:

UMCP is the first HTN planning formalism to be provably sound and complete. There are other HTN formalisms presented in the planning literature. This section summarizes some of these formalisms.

- Tate [47] introduced several notable techniques in his planning system Nonlin. Previously, Sacerdoti's NOAH [41] had decomposition schemas that were written using a procedural language. In Nonlin, decomposition schemas are specified using a declarative language. He defined three types of conditions which the planner can use for different purposes: use-when conditions are to be used to filter out decomposition schemas inapplicable to the problem; supervised conditions specify conditions which must be made true by the planner; and unsupervised conditions indicate that other parts of the plan would satisfy the conditions. Some of these condition types are described later in Chapter 4, where they are compared to a new condition type. He also introduced 'causal links', as described in the previous section, as a way to keep track of which step establishes which condition.

  Following Nonlin, two general-domain HTN planners were developed. SIPE [54] and O-Plan [10] employ many techniques similar to the ones used in Nonlin. They were developed with real-world planning applications in mind. Thus, they have extensive capabilities to deal with real-world requirements. Both of them are capable of execution monitoring as well as plan generation. Both have added more condition types to specify a domain, thus allowing domain experts more control of the search.

- Yang [55] used a rather restrictive hierarchical planning formalism in order to satisfy conditions in a plan at higher levels. He analyzed that in some situations, conflicts in a plan can only be resolved by further decomposing non-primitive actions in the plan. However, since there is no guarantee that the conflicts can be resolved at the primitive level, such decompositions may turn out to be redundant. He suggested, by modifying decomposition schemas in a certain way, the planner can improve its planning efficiency by avoiding such redundant decompositions.

- In their DPOCL formalism, Young, et al. [57] present a way to combine causal planning and HTN planning which include action-based operators that can be used to establish un-satisfied conditions in a plan and non-primitive tasks that must be decomposed by applying decomposition schemas. At each cycle, the DPOCL algorithm non-deterministically chooses between causal planning and decompositional planning. Although their restrictions on the preconditions and effects of non-primitive tasks are less strict than the ones used by Yang described above, it still does not allow users to specify some conditions, such as "user intent" conditions [24] that can be specified in UMCP.

- Kambhampati, et al. [24] also present a planning formalism for partially hierarchical domains. The combination of HTN and action-based planning allows the planner to plan both by task decomposition and condition satisfaction. Unlike DPOCL where both abstract actions and primitive actions are tried in order to establish a goal, Kambhampati, et al. use dominance relations between actions when determining which actions to use to establish a goal. An action t is *dominated* by another action t' if there exists a decomposition of t' that contains t. If both t and t' establish the same goal and t is dominated by t', then t will not be used to establish the goal. This method ensures the systematicity of the algorithm.

2.3.5 Comparison between HTN and action-based planning

There are many differences in HTN planning and action-based planning. The following are some discussions of advantages and disadvantages of HTN planning and action-based planning from various points.

- **Expressivity:** Erol, et al. [12], in their formal analyses of HTN planning, have shown that HTN planning is provably more expressive than action-based planning using the analogy with context-free grammar [12].

  Compared with action-based operators where users can specify domain restrictions only in the preconditions and effects of atomic actions, HTN planning operators allow users to specify many more types of domain restrictions. Thus, HTN operators have ways to filter out undesirable plans that action-based operators do not. For example, in order to get money, an action-based planner may generate a plan which opens an account at a certain bank in order to use a specific ATM. This may be a feasible plan and its operational cost can be lower than a plan which directs you to an ATM further away, except many of us do not want to have too many bank accounts. One can easily specify conditions to prune out such feasible-but-not-realistic plans in HTN operators.

- **Efficiency:** HTN planning is capable of using more domain knowledge to guide the search. However, there have been no extensive comparisons between the efficiencies of HTN planning and action-based planning, although the experimental results by Barrett and Weld [5] suggests that HTN planning is faster than classical action-based planning.

- **Domain Modeling:** Action-based operators can only be used for condition-goals (i.e. conditions which must be true after the execution of a plan), while HTN planners can plan not only for condition-goals but also for action-goals (i.e. abstract actions that needs to be performed). While action-goals (do-X) can be expressed as condition-goals (X-done), such a roundabout way of specifying domains is often not desirable.

  One of the difficulties domain experts face when modeling their domain in HTN planning is that they need to decide what kind of abstract tasks they want to specify in advance, which may not be quite clear. Ideally, abstract tasks should be specified such that they are intuitive to users yet efficient for the planner. Hybrid algorithms such as the one by Kambhampati, et al. [24] that allow gradually building the hierarchy of a domain may help domain experts specify abstract tasks wisely.

- **Extensibility:** One important aspect of a planning formalism is extensability. Many real-world problem domains require abilities to do temporal reasoning and numeric calculations while other applications require interleaving planning and execution. HTN planning formalism makes it easier to work on some parts of a plan which do not affect the entire plan. Such changes are quite difficult to be integrated in action-based planning. For example, Smith, et al. [44] have used HTN planning techniques for their computer bridge playing program in order to reason about possible moves by other players in addition to the moves the program can make.

Chapter 3

Fewest Alternative First Method

As mentioned in the previous chapter, one popular refinement strategy is some type of least commitment 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. One reason why the least-commitment idea is 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 “least commitment” idea to the same kind of refinement throughout the entire planning process. In order to do planning at all, a planner has to refine something—and thus, when a planner postpones refining one aspect of the plan it is generating, this may make it prematurely refine some other aspects 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. This strategy is called the "fewest alternatives first" (FAF) strategy. For partial-order planning with STRIPS-style operators, Joslin and Pollack [20, 21] found that a version of this strategy outperformed the "least commitment to open conditions" strategies. The same heuristic as FAF was used in constraint satisfaction problems as early as 1975 [6] as a search rearrangement method and Purdom [40] analyzed its application to SAT problems. Although the heuristic has been used in planning for some time [10], there was little analysis on how the heuristic is effective in AI planning.

In this chapter, the FAF heuristic is addressed from various aspects. First, a version of the FAF heuristic is compared with a "least commitment to task achievement" strategy and a "least commitment to variable bindings" strategy in HTN planning. The experimental results show that the version of FAF does as well as or better than either one of the two least commitment strategies. In Section 3.2, refinement strategies are modeled as methods to serialize AND/OR graphs. The efficiency of a strategy can then be measured by the size of the serialized graph. The FAF heuristic is compared with the best and worst serializations theoretically and empirically. Section 3.3 presents related work and Section 3.4 summarizes the results.

3.1 Commitments to Variable Bindings

Many planners use a "least commitment to variable bindings" strategy where variable instantiation is delayed until it is necessary. The experimental results shown by Yang and Chan [56] seem to imply that their "least commitment to variable bindings" strategy always performs better than (or at least as well as) the default strategy of the SNLP planner in which a variable is instantiated with a constant value when there is a condition on the variable.

This section investigates the effects of variable commitments for HTN planning, to see if it also has a strong performance effect, as indicated by the results of Yang and Chan in action-based planning. Many planners, including UMCP, keep a set of possible values for each variable. When enforcing a constraint that affects a variable, the planner tries to constrain the set of possible values for the variable instead of immediately instantiating it. For example, suppose a condition (*p* x) will be satisfied in the specified world state if the value of x is a, b or c. the planner creates a partial plan where x has the possible value set {a, b, c}, instead of creating three partial plans with alternate variable assignments (i.e. x = a, x = b or x = c). Such a refinement strategy helps to keep the size of the search space small by avoiding unnecessary branching in the search. However, some constraints that involve two or more variables cannot be enforced without branching. For example, suppose in the current partial plan P, the possible values for x and the possible values for y are both {a, b}. Then in order to enforce the non-codesignation constraint ($x \neq y$), the planner has to create two partial plans, one with x = a and y = b, and another with x = b and y = a. Another example would be enforcing the state constraint (before (*color* ?ball ?color) n), when both the conditions (*color* ball1 blue) and (*color* ball2 green) are true in the state just before the task referred to by n. In either case, the planner can enforce the constraint by instantiating the variables involved, resulting in two partial plans. Upon encountering such constraints, a planner has to determine whether to enforce them now or delay the enforcement. Some planners delay the enforcement of the constraints until the time when the variables are constrained enough (by some other constraints) for the planner to avoid branching, or until the time when there are no other refinements applicable to the partial plan. The UMCP planner delays the enforcement of non-codesignation constraints but enforces state constraints immediately if they are enforceable in the current partial plan. The problem with the delaying method is that the deferred constraints may not be establishable, and in that case delaying the enforcement of constraints could incur a large backtracking cost.[1] On the other hand, additional constraints may constrain variables in a way the planner can avoid unnecessary branching. Consider the above example of the state constraint (before (*color* ?ball ?color) n) again. If the planner later encounters a constraint that binds the variable ?color to green, then it can satisfy the condition by binding the variable ?ball to ball2. Since it is difficult for the planner to foresee what types of constraints it will encounter later, it needs some heuristic to aid the decision process.

We compared the relative performance of three variable commitment strategies for HTN planning: the Reluctant Variable Binding Strategy (RVBS), which does least commitment to variable bindings, the Eager Variable Instantiation Strategy (EVIS), in which no non-primitive task is expanded until all variable constraints are committed; and the Dynamic Variable Commitment Strategy (DVCS), a FAF strategy which chooses between expansion and variable instantiation based on the number of branches that will be created in the search tree. The results show that there are planning domains in which EVIS does well, and planning domains where it does poorly. The same is true for RVBS. However, DVCS, which can choose between eager variable commitment and reluctant variable commitment depending on what looks best for the task at hand, does well over a broader range of planning domains.

[1]This problem is also investigated in [21] using action-based planning framework.

3.1.1 Strategies

As argued above, avoiding refinements to one aspect of planning can lead to premature refinements to other aspects of planning. Thus, what a planner needs is some way to find a balance between the refinements of different aspects of planning. One way is to use the FAF heuristic which chooses a refinement which generates fewer immediate search nodes. We created an implementation of such a "dynamic commitment" strategy and compared it experimentally with implementations of a "least commitment to variable bindings" strategy and a "least commitments to task instantiation" strategy. More specifically, the commitment strategies are as follows:

- Eager Variable Instantiation Strategy (EVIS). This is an HTN version of the eager variable commitment strategy. When there are constraints on variables in the current partial plan, don't expand any non-primitive task until all constraints are satisfied. Instantiate variables into constants whenever necessary to resolve constraints.

- Reluctant Variable Binding Strategy (RVBS). This is basically the opposite strategy. Delay instantiating variables as much as possible. If a constraint cannot be satisfied without generating more than one partial plan, expand all tasks before committing to such constraints.

- Dynamic Variable Commitment Strategy (DVCS). This strategy attempts to minimize the branching factor as discussed earlier. Suppose T is the current partial plan and there is a constraint C in T which cannot be satisfied without generating more than one partial plan. For each variable x in C, let v(x) be the number of possible values for v; and for each task t in T, let m(t) be the number of methods that unify with t. Let V = min v(x) : x is a variable in C; and let M = min m(t) : t is a task in T. If V < M, then choose to instantiate the variable x for which v(x) is smallest. If M $\leq$ V, then choose to expand the task t for which m(t) is smallest. Although this decision criterion may seem more complicated than EVIS and RVBS, the overhead involved in computing it is negligible. When M = V, expansions are favored over instantiations because further constraint refinements might constrain the possible value set but not limit the number of methods. Unless the task network is pruned, expansion will eventually take place with the same number of methods. On the other hand, it is possible to instantiate a variable with fewer possible values if the instantiation is delayed.

To make the comparison between strategies easier, RVBS and EVIS use the same selection methods that DVCS uses to choose tasks and variables. More specifically, when the strategy decides to expand a task, it expands the task t with the minimum number of methods in the current partial plan. When the strategy decides to instantiate a variable to satisfy a constraint C, it instantiates the variable v in C that has the minimum number of possible values.

3.1.2 Experiments

We compared the EVIS, RVBS, and DVCS refinement strategies by using them in the UMCP planner on randomly chosen problems in three different planning domains. The three planning domains—and the experimental results in those domains—are described below. The experiments were run using Allegro Common Lisp on a SUN Sparc station, and running UMCP1.0 with a depth-first search strategy. For each problem and each refinement strategy, both the CPU time and

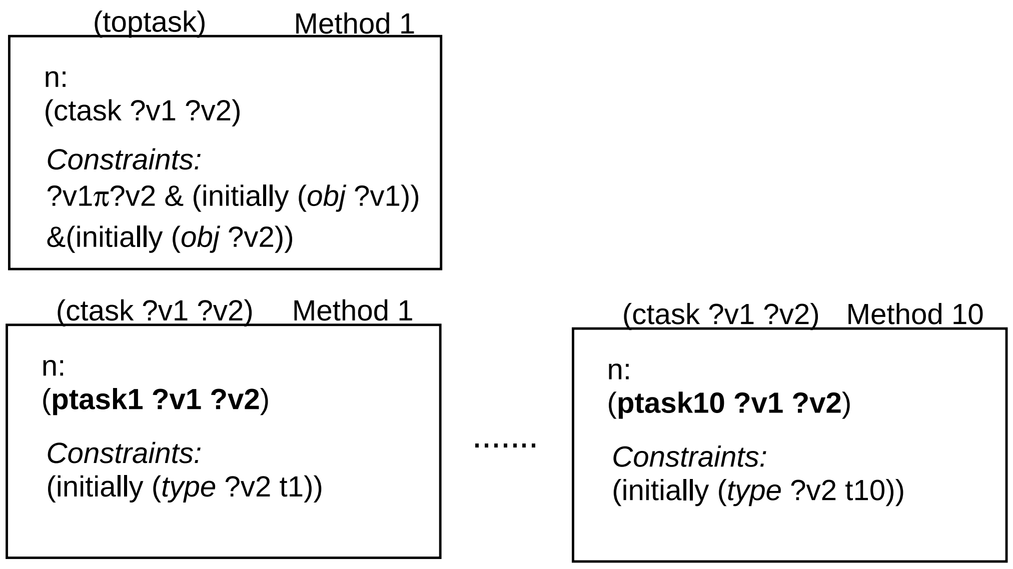


Figure 3.1: Methods for Domain A

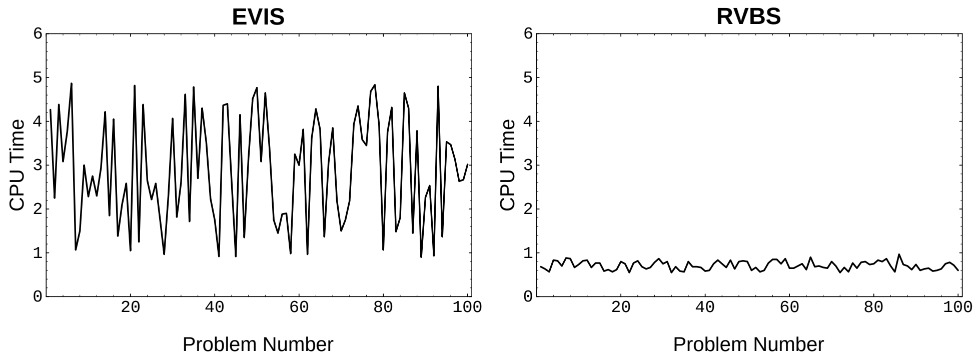
the number of nodes (i.e., the number of task networks) generated were counted. Since both measurements gave similar results, only the CPU time is shown.

3.1.3 Domain A

In Domain A the goal is to find a way to accomplish a 0-ary task (toptask). As shown in Figure 3.1, (toptask) expands into a 2-ary task (ctask ?v1 ?v2), where ?v1 and ?v2 are variables; and there are ten different methods for expanding (ctask ?v1 ?v2). The initial state is the set

$$\{ \text{ (}\mathit{obj}\text{ obj1), (}\mathit{obj}\text{ obj2), } \cdots \text{, (}\mathit{obj}\text{ obj10), (}\mathit{type}\ o\ t\text{) } \}$$

where $o \in$ {obj1, $\cdots$, obj10} and $t \in$ {t1, $\cdots$, t10}. Different planning problems are specified by choosing different values for o and t. Since the initial state has exactly one type literal, there is only one successful way to bind the variable ?v2 and expand the task (ctask ?v1 ?v2). The planning problem is to find the method



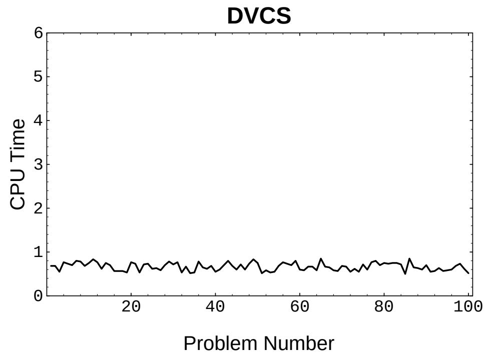


Figure 3.2: CPU time (in seconds) in Domain A

that works. EVIS, RVBS, and DVCS were compared in Domain A by running them on a suite of 100 randomly generated problems. Figure 3.2 shows the performance of UMCP with the three commitment strategies. There is exactly one solution for each problem. For each problem, RVBS and DVCS always find this solution after creating 14 task networks. Depending on the problem, EVIS creates between 24 and 114 task networks. UMCP's average CPU times were 2.88 seconds using EVIS, 0.71 seconds using RVBS, and 0.66 seconds using DVCS.

EVIS has more trouble than RVBS and DVCS because it instantiates the variable ?v2 before expanding the task ctask, and this tends to bind ?v2 to an object that does not meet the constraint found in the methods of ctask. On the other hand, RVBS does not instantiate ?v2 until after enforcing the constraint (*type* ?v2 t) so it does not make an instantiation of ?v2 which eventually fails. DVCS chooses to expand ctask before the instantiation of ?v2 since the values of V and M are the same (10), and thus performs identically to RVBS.

3.1.4 Domain B

Domain B is basically an encoding of the well known arc-consistency problem [27]. As in Domain A, the goal is to accomplish toptask; but the methods are different. As shown in Figure 3.3, toptask expands into ctask1, ctask1 expands into ctask2, and ctask2 expands into ctask3. The methods for ctask1 specify that ?v1, ?v2 and ?v3 must have different values but the same type. ctask2 and ctask3 each have four identical methods, which increases the branching factor when UMCP does task expansion. The initial state is the set

$$\{\ (\textit{obj}\ \text{obj1}), (\textit{obj}\ \text{obj2}), \cdots, (\textit{obj}\ \text{obj7}),$$
$$(\textit{type}\ \text{obj1}\ t_1), (\textit{type}\ \text{obj2}\ t_2), \cdots, (\textit{type}\ \text{obj7}\ t_7)\},$$

where each t_i is t1, t2 or t3. Different planning problems in this domain are specified by choosing different values for each of the t_i. The problem is to find three different objects which share the same object type. In Domain B, a suite of 50 problems was created by randomly assigning types to each object obj_i in the initial state. Each problem had at least one solution. The results are shown in Figure 3.4. EVIS and DVCS created the same number of task networks for each test problem, and incurred about the same amount of CPU time. UMCP averaged 1.09 seconds and 1.10 seconds, respectively. RVBS never did better than

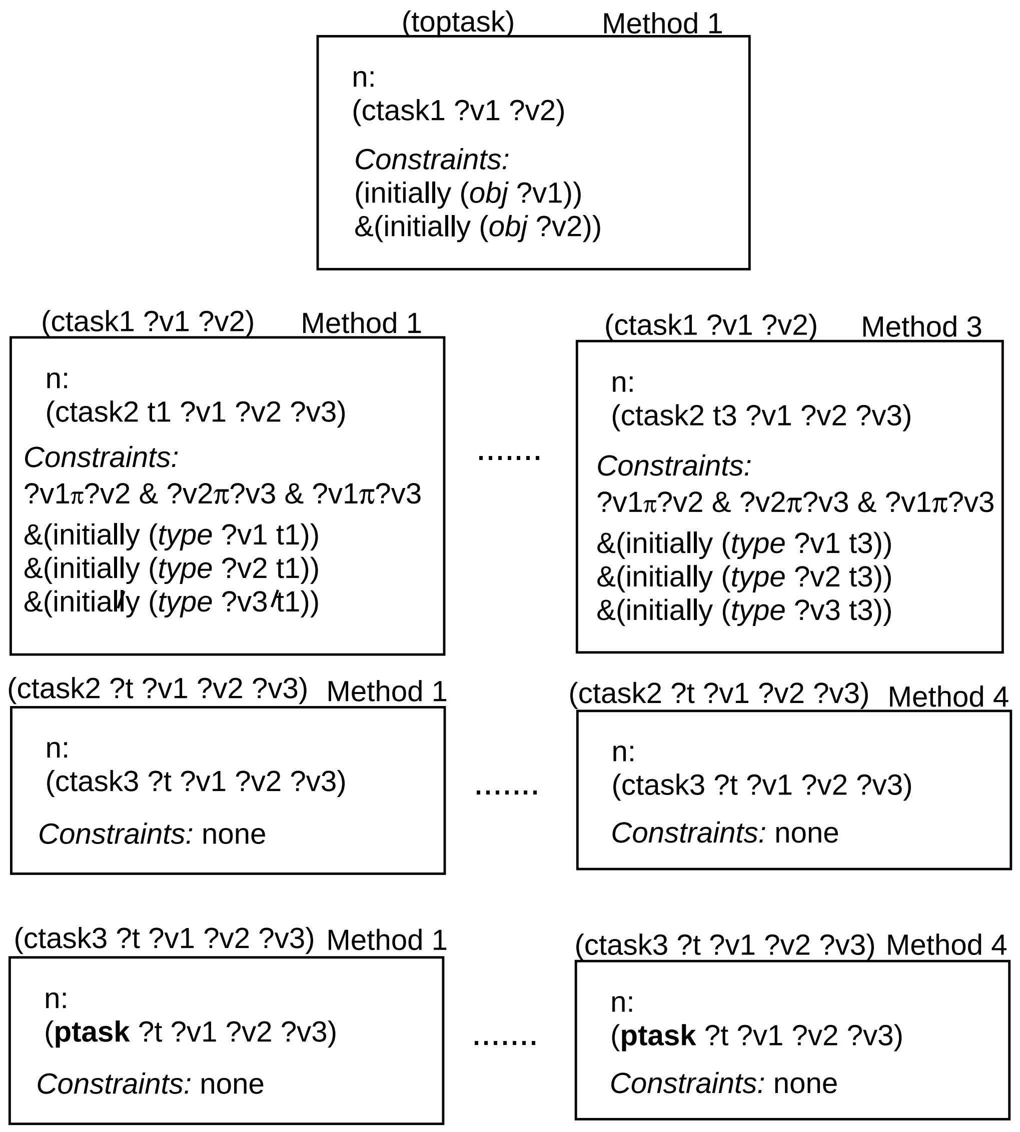

(toptask) Method 1
n:
(ctask1 ?v1 ?v2)
Constraints:
(initially (obj ?v1))
&(initially (obj ?v2))
(ctask1 ?v1 ?v2) Method 1
n:
(ctask2 t1 ?v1 ?v2 ?v3)
Constraints:
?v1π?v2 & ?v2π?v3 & ?v1π?v3
&(initially (type ?v1 t1))
&(initially (type ?v2 t1))
&(initially (type ?v3 t1))
....
(ctask1 ?v1 ?v2) Method 3
n:
(ctask2 t3 ?v1 ?v2 ?v3)
Constraints:
?v1π?v2 & ?v2π?v3 & ?v1π?v3
&(initially (type ?v1 t3))
&(initially (type ?v2 t3))
&(initially (type ?v3 t3))
(ctask2 ?t ?v1 ?v2 ?v3) Method 1
n:
(ctask3 ?t ?v1 ?v2 ?v3)
Constraints: none
....
(ctask2 ?t ?v1 ?v2 ?v3) Method 4
n:
(ctask3 ?t ?v1 ?v2 ?v3)
Constraints: none
(ctask3 ?t ?v1 ?v2 ?v3) Method 1
n:
(ptask ?t ?v1 ?v2 ?v3)
Constraints: none
....
(ctask3 ?t ?v1 ?v2 ?v3) Method 4
n:
(ptask ?t ?v1 ?v2 ?v3)
Constraints: none

Figure 3.3: Methods for Domain B

EVIS or DVCS, and usually did much worse. On the average, UMCP's CPU time with RVBS was 2.54 seconds. The reason for these results is that when ctask1 is decomposed by using a particular method, each variable ?v1, ?v2 or ?v3 has to be an object that is of the particular type. If there are only two or less obj_i's in the initial state that are of the particular type, the partial plan will fail. Thus, it is better to instantiate the variables if the variable has two or less possible values. Thus, since EVIS instantiates variables ?v1, ?v2 and ?v3 before expanding the task ctask2, EVIS can prune the partial plans which cannot satisfy the constraints imposed in the methods for ctask1. Also, since DVCS chooses to instantiate variables when they have three or less possible values, rather than decomposing ctask2, DVCS also can prune such partial plans. On the other hand, RVBS does not instantiate variables until they are fully expanded into primitive task networks. Thus RVBS generates task networks that would not be generated by EVIS.

3.1.5 Domain C

As shown in Figures 3.5 and 3.6, Domain C contains tasks and methods similar to those from both Domains A and B. Solving the problem involves combining methods similar to those in Domain A with methods similar to those in Domain B—but the order in which these methods should be used depends on whether the goal is toptaska or toptaskb. The initial state contains the atoms

$$\{\ (\textit{obj}\ \text{obj1}), (\textit{obj}\ \text{obj2}), \cdots, (\textit{obj}\ \text{obj10})\ \},$$

and also fifteen atoms of the form $(type\ o\ t)$ where $type \in \{\textit{type1,type2}\}$; $o \in \{$obj1, ..., obj10 $\}$; and $t \in \{$t1,t2,t3$\}$. Different planning problems are specified by choosing different values for o and t, as well as by choosing either toptaska or toptaskb as the

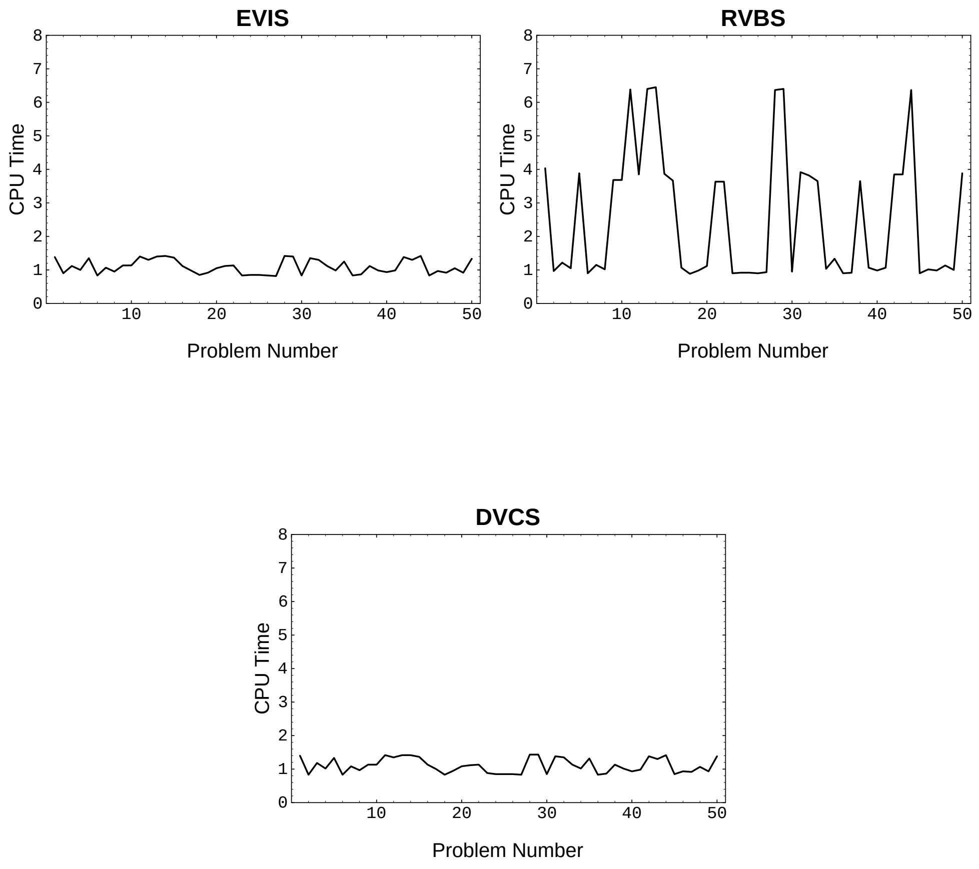


Figure 3.4: CPU time (in seconds) in Domain B

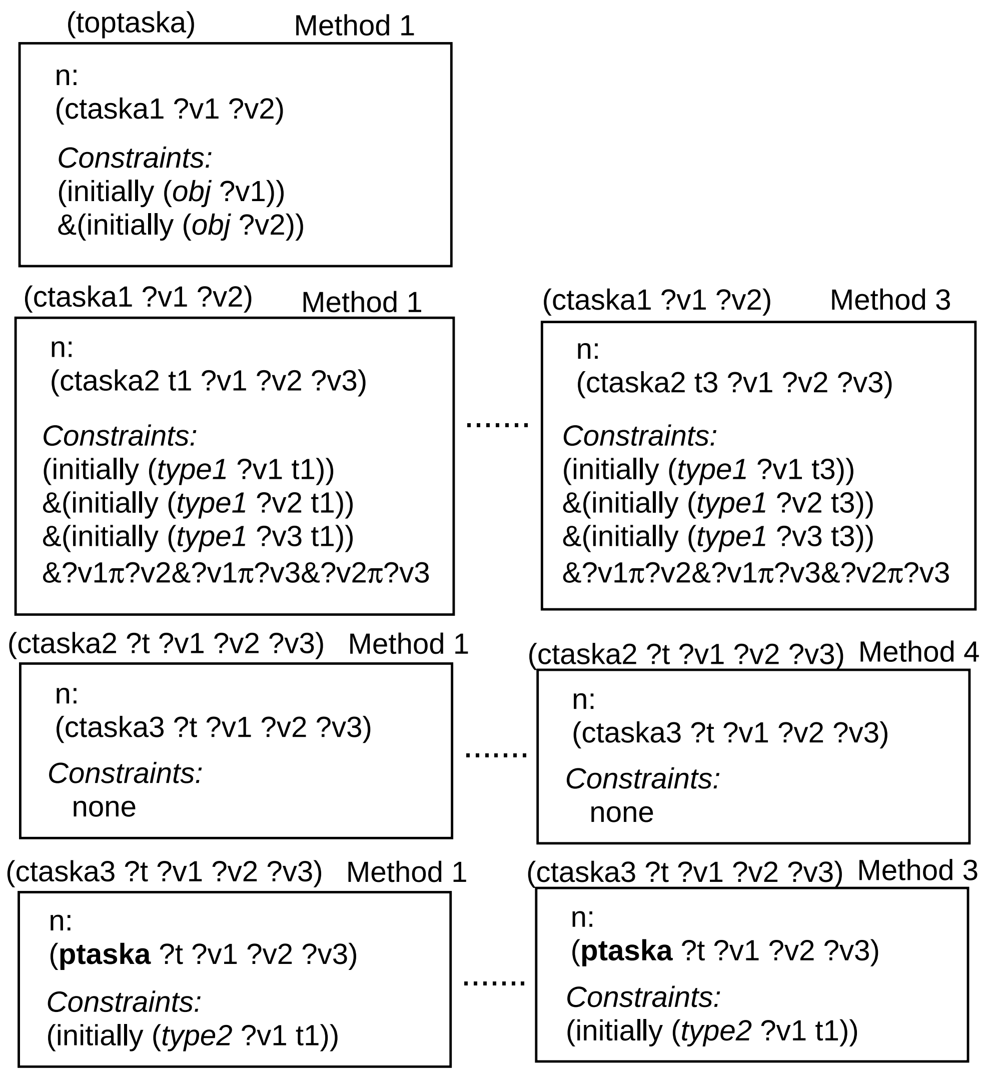


Figure 3.5: Methods for the decomposition of (toptaska) in Domain C

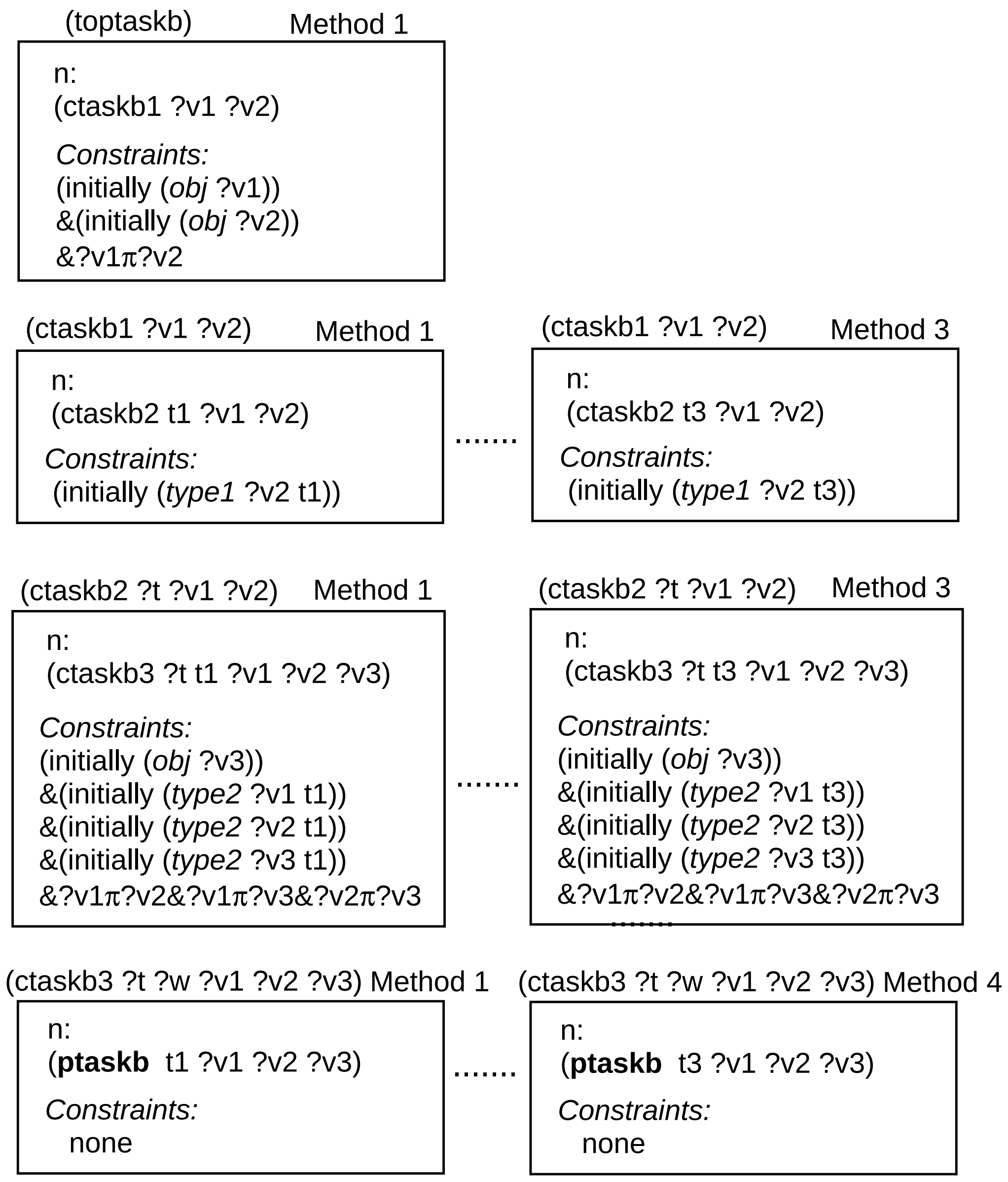

(toptaskb) Method 1
n:
(ctaskb1 ?v1 ?v2)
Constraints:
(initially (obj ?v1))
&(initially (obj ?v2))
&?v1π?v2
(ctaskb1 ?v1 ?v2) Method 1
n:
(ctaskb2 t1 ?v1 ?v2)
Constraints:
(initially (type1 ?v2 t1))
....
(ctaskb1 ?v1 ?v2) Method 3
n:
(ctaskb2 t3 ?v1 ?v2)
Constraints:
(initially (type1 ?v2 t3))
(ctaskb2 ?t ?v1 ?v2) Method 1
n:
(ctaskb3 ?t t1 ?v1 ?v2 ?v3)
Constraints:
(initially (obj ?v3))
&(initially (type2 ?v1 t1))
&(initially (type2 ?v2 t1))
&(initially (type2 ?v3 t1))
&?v1π?v2&?v1π?v3&?v2π?v3
....
(ctaskb2 ?t ?v1 ?v2) Method 3
n:
(ctaskb3 ?t t3 ?v1 ?v2 ?v3)
Constraints:
(initially (obj ?v3))
&(initially (type2 ?v1 t3))
&(initially (type2 ?v2 t3))
&(initially (type2 ?v3 t3))
&?v1π?v2&?v1π?v3&?v2π?v3
(ctaskb3 ?t ?w ?v1 ?v2 ?v3) Method 1
n:
(ptaskb t1 ?v1 ?v2 ?v3)
Constraints:
none
....
(ctaskb3 ?t ?w ?v1 ?v2 ?v3) Method 4
n:
(ptaskb t3 ?v1 ?v2 ?v3)
Constraints:
none

Figure 3.6: Methods for the decomposition of (toptaskb) in Domain C

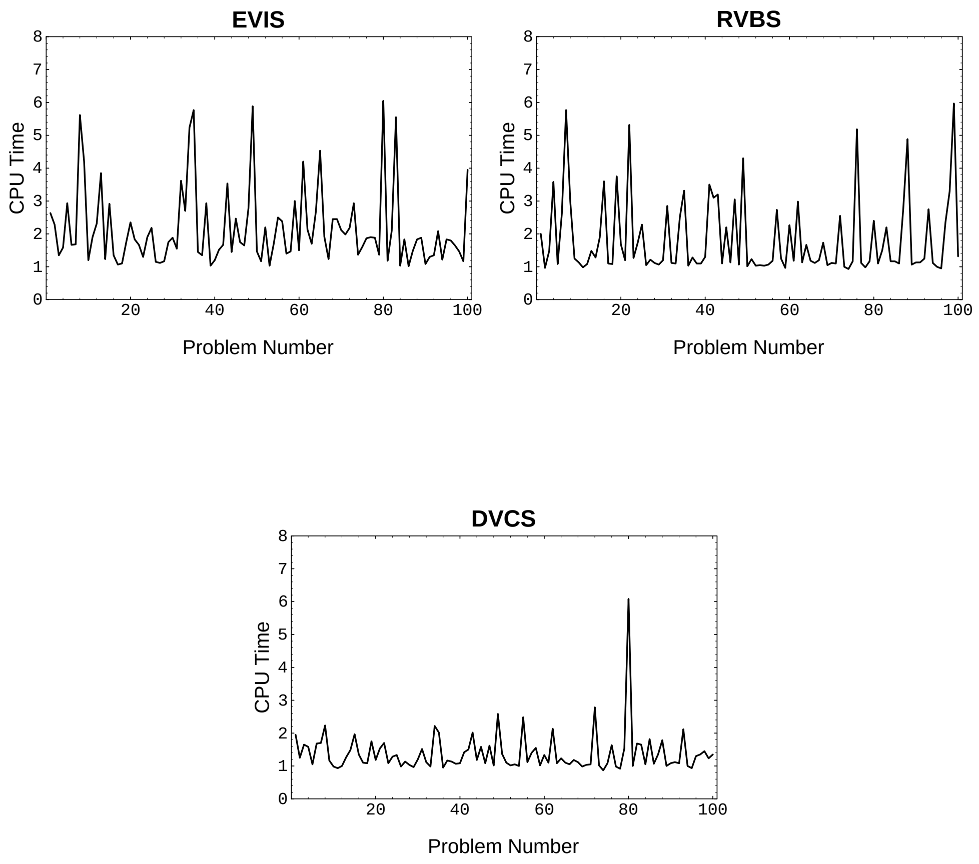


Figure 3.7: CPU time (in seconds) in Domain C

goal. In Domain C, a suite of 100 problems were created by randomly selecting the goal tasks and initial states. Of these problems, 44 problems had the goal task toptaska and 56 problems had the goal task toptaskb. Seven of the 100 problems had no solutions. As shown in Figure 3.7, DVCS had the best performance overall. UMCP's average CPU times were 2.15 seconds using EVIS, 1.83 seconds using RVBS, and 1.38 seconds using DVCS.

T-test

To test whether or not the differences shown in Figure 3.7 were statistically significant, a paired sample t-test was applied to the results. Let μ_D be UMCP's mean CPU time using DVCS and μ_R be UMCP's mean CPU time using RVBS. The null hypothesis H_0 is that $\mu_R - \mu_D = 0$ (or H_0: $\mu_R = \mu_D$); the alternative hypothesis H_1 is that $\mu_R - \mu_D > 0$. The t statistic computed from the results is 5.569. This is greater than the value 2.626 of the t-distribution with probability 0.995 where the degrees of freedom is 100. Thus we can reject H_0 and say that the difference of the means is significant. Similarly, the difference of the mean CPU time for DVCS and the mean CPU time for EVIS is significant with the t statistic 8.155 and the confidence level greater than 0.999. The reason why DVCS outperformed EVIS and RVBS is that even while solving a single planning problem, which refinement strategy is best can vary from task to task—and DVCS can select between the EVIS and RVBS strategies on the fly.

3.1.6 Weighed DVCS

The above experimental results show that neither RVBS nor EVIS performs well for all kinds of domains. The DVCS strategy can outperform RVBS and EVIS by alternating between the two strategies. The selection of a strategy in DVCS is made by calculating the number of immediate search nodes each strategy generates and choosing the strategy which generates fewer search nodes. The question remains that if this method of selection is the best selection. In order to investigate the strategy selection methods, We modified DVCS to create a weighed DVCS strategy (WDVCS) which takes a value r ($0 \leq r \leq 1$) to put weights on EVIS and RVBS when selecting a strategy. More specifically, WDVCS chooses a refinement

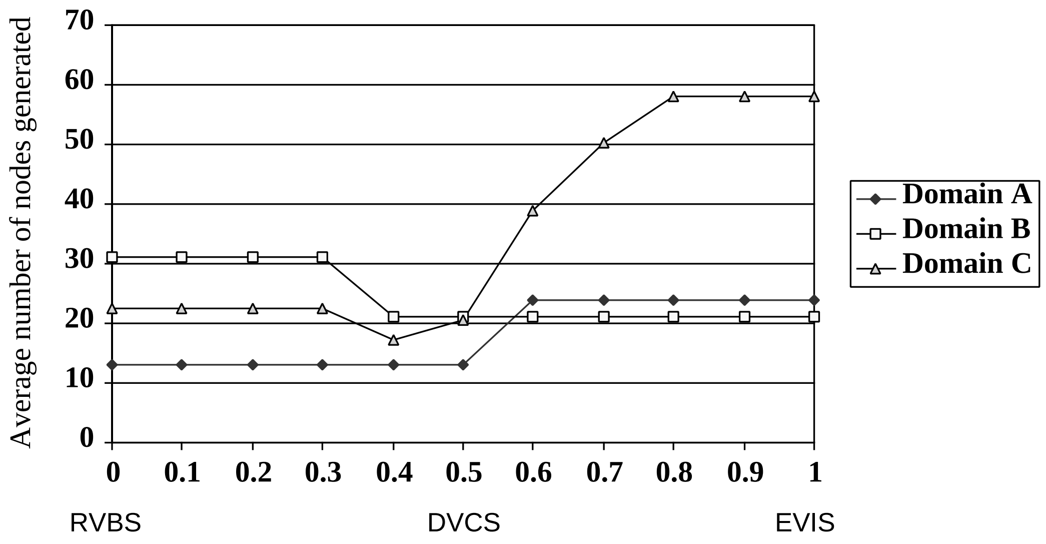


Figure 3.8: The results for the problems in Domains A, B and C by applying the WDVCS strategies. The x-axis shows the r-value and the y-axis shows the average number of nodes generated.

as follows: For each variable x in T, let v(x) be the number of possible values for v; and for each task t in T, let m(t) be the number of methods that unify with t. Let V = min v(x) : x is a variable in T; and let M = min m(t) : t is a task in T. If (1-r)*V < r*M, then choose to instantiate the variable x for which v(x) is smallest. If (1-r)*V $\geq$ r*M, then choose to expand the task t for which m(t) is smallest. Thus, WDVCS simulates RVBS when r = 0, DVCS when r = 0.5 and EVIS when r = 1.

Figure 3.8 shows the results of WDVCS on Domains A, B and C problems that were generated in the experiments of RVBS, EVIS and DVCS. The r-value is varied from 0 to 1 with an increment of 0.1. The results show that WDVCS does best when r $\leq$ 0.5 on Domain A, when r $\geq$ 0.4 on Domain B, and when r = 0.4 on Domain C. This suggests that DVCS is not always the best strategy to choose between the two strategies, although we suspect that the WDVCS strategy with r = 0.4 does not perform best in all types of domains.

3.1.7 Summary

Many planning papers are devoted to various least commitment strategies which postpone some types of refinements during planning process. However, least commitment does not necessarily do well on all kinds of problems. In order to plan, a planner needs to make decisions; if it postpones some type of refinements, it has to work on other types of refinements. Thus, a planner cannot apply least commitments to all aspects of planning.

Three strategies were tested: EVIS and RVBS are two least commitment strategies where each tries to delay different type of commitments. EVIS uses a "least commitment to task instantiation" strategy, while RVBS uses a "least commitment to variable bindings" strategy. DVCS uses the FAF heuristic to dynamically switch between EVIS and RVBS. The experiments on Domains A and B showed that neither one of the two strategies can perform well on all kinds of problems. The third strategy, DVCS, uses the FAF heuristic to switch between EVIS and RVBS. More specifically, DVCS chooses to work on the refinement which generates fewest immediate search nodes. On Domain A, where RVBS outperformed EVIS, DVCS performed as well as RVBS. On Domain B, where EVIS outperformed RVBS, DVCS performed as well as EVIS. And on Domain C, which is a combination of the Domains A and B, DVCS outperformed either EVIS or RVBS for most of the problems.

Although the three test domains contain only non-codesignation constraints as constraints that cannot be fully established without branching, similar performance of the three variable commitment strategies can be expected on the domains that contain state constraints such as the one described at the beginning of this section.

The performance of a variable commitment strategy depends on many factors including:

- The probability of a variable instantiation succeeding: If a variable has many values the variable can be instantiated with, but only one value leads to a solution plan, then a strategy that delays instantiating the variable performs better than a strategy that eagerly instantiates variables. On the other hand, the opposite is true if a variable may have many values that the variable can be instantiated with and each value heads to a solution plan.

- The probability of a decomposition method succeeding: Like variable instantiations, a task may have many decomposition methods but only one of them may work in the current problem, or all of them may succeed in the current problem.

- Existence of inconsistent variable bindings: As in Domain B, some problems may make the planner handle constraints that do not have a consistent set of variable bindings. In order to find such inconsistency, a planner can instantiate variables, as EVIS does, or run a constraint satisfaction problem solver, as FSNLP [56] does. Either way, if there is no inconsistency, these operations could impose big CPU time.

Also, many of these factors can influence each other. For example, choosing some decomposition method may add new constraints to the partial plan that make the variable bindings in the plan inconsistent. Considering all the factors, a dynamic variable commitment strategy that switches between different strategies depending on the current situation is necessary to maximize the efficiency of variable commitments in planning.

We also tested the WDVCS strategy which switches between EVIS and RVBS with varying amount of weight for each strategy. DVCS can be viewed as the WDVCS strategy which put the same amounts of weights to both EVIS and RVBS. The experimental results on WDVCS show that DVCS is not the best strategy for all of the domains; on Domain C, WDVCS performed best when the r-value is 0.4 and the RVBS is slightly more preferred than EVIS. However, we doubt that WDVCS with the r-value = 0.4 always performs best in all types of problems. Rather, WDVCS would probably perform best with different r-values in different sets of problems, depending on the factors listed above. However, the results strongly imply the best strategy lies in somewhere in the middle, considering the worst performance in each domain appears at the end (i.e. r-value = 0 or 1). Thus, DVCS seems to be the best bet for many types of problems.

3.2 Graph Serialization

The experimental results shown in the previous section seems to indicate that the FAF heuristic performs well in choosing between task instantiations and variable bindings. This section examines the FAF strategy in more detail, to try to understand whether it can be expected to perform well in general—and if so, then why.

The search process that is carried out by an AI planning system can be seen as taking an AND/OR graph and generating from it an equivalent state-space graph, one OR-branch at a time. This process is called serializing the AND/OR graph.

Different refinement strategies for planning correspond to different strategies for serializing the AND/OR graph. Since different serialization strategies produce different search spaces, they contain different numbers of nodes. This section analyzes the size of search space the FAF strategy by looking at the the sizes of serialized trees that can be obtained by applying the FAF strategy to serialize various AND/OR graphs.

3.2.1 Partial-Order Planning and AND/OR Graphs

The space searched by a partial-order planner may be thought of as an AND/OR graph in the following manner:

- Given a partially developed plan, there may be several elements of the plan that need to be refined in one way or another. These could 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.

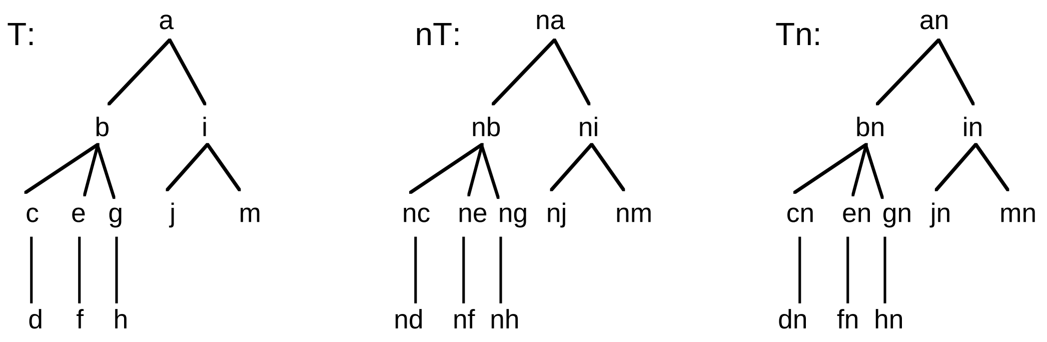


Figure 3.9: A tree T, and the trees nT and Tn (where n is a node not in T).

If the refinements performed on a plan were independent in their effects on the plan, a partial-order 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, partial-order 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 search tree 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 a formal definition is given only for the special case where the AND/OR graph is binary (i.e., each non-leaf node has exactly two children). However, it should be obvious to the reader how to generalize this definition for the non-binary case.

First, look at the following notation (see Figure 3.9 for examples).

If T is a tree whose node set is N and whose edge set is E; and n is any node not in N, then:

- Tn is the tree whose node set is $\{np : n \in N\}$ and whose edge set is $\{(mp, np) : (m, n) \in E\}$;
- nT is the tree whose node set is $\{pn : n \in N\}$ and whose edge set is $\{(pm, pn) : (m, n) \in E\}$.

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 emanating from G's root node g is a binary OR-branch. Let H and I be the AND/OR graphs rooted at the two children of g; and let S and T be any serializations of H and I, respectively. Then as shown in Figure 3.10, the tree R whose root is g and whose subtrees are S and T is a serialization of G.

Case 3: G's root node g is not a leaf, and the branch emanating from g is a binary AND-branch. Let H and I be the AND/OR graphs rooted at the two children of g. Let S be any serialization of H, and let T be any serialization of I. Let S's root be s and its leaf nodes be s_1, s_2, ..., s_p; and let T's root be t and its leaf nodes be t_1, t_2, ..., t_q. Then as shown in Figure 3.11, 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, ..., s_pt the trees s_1T, s_2T, ..., s_pT, respectively;
  - the tree R_2 formed by taking the tree sT, and attaching to its leaves st_1, st_2, ..., st_q the trees St_1, St_2, ..., St_q, respectively.

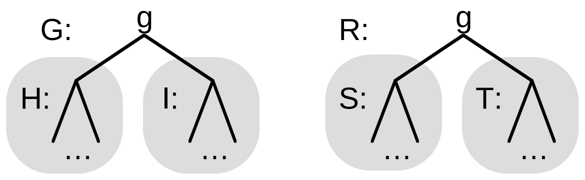


Figure 3.10: Case 2 of serializing an AND/OR graph.

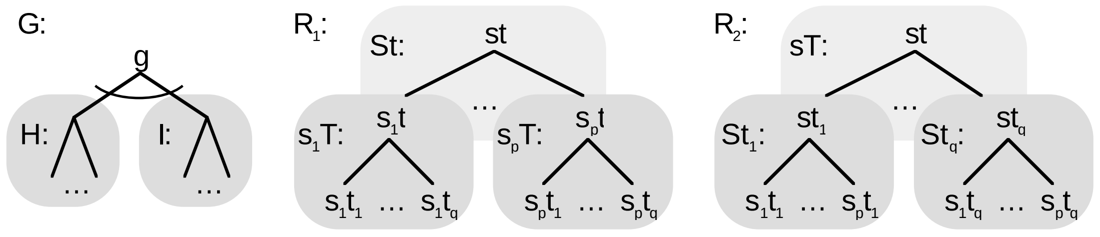


Figure 3.11: Case 3 of serializing an AND/OR graph.

In Figure 3.11, 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 3.12 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 [46]. One well known example is Prolog's search procedure (for example, see Clocksin and Mellish [9]), which serializes AND/OR graphs in a depth-first left-to-right manner. For example, in graph G of Figure 3.12, suppose that each node corresponds to a logical atom, each AND-branch corresponds to a Horn clause, and each OR-branch corresponds

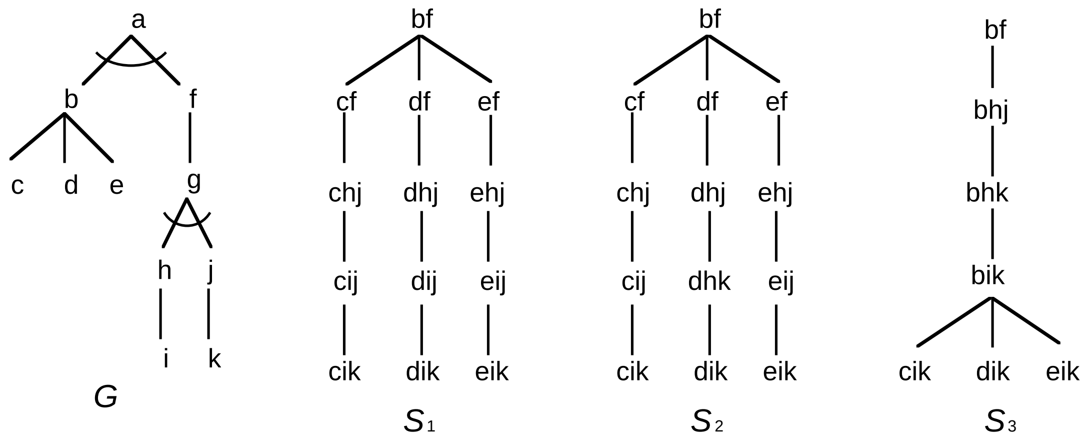
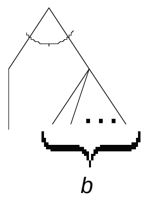


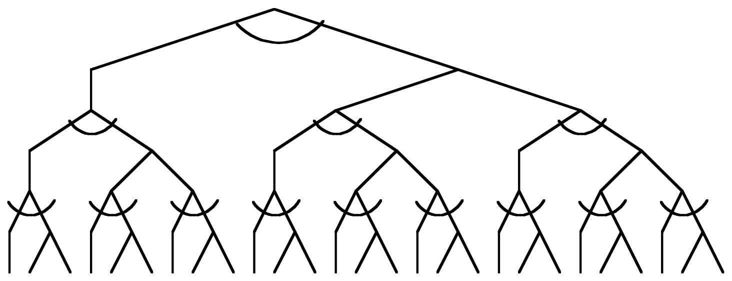
Figure 3.12: A simple AND/OR graph G, and three serializations S_1, S_2, and S_3.

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 3.12. 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.

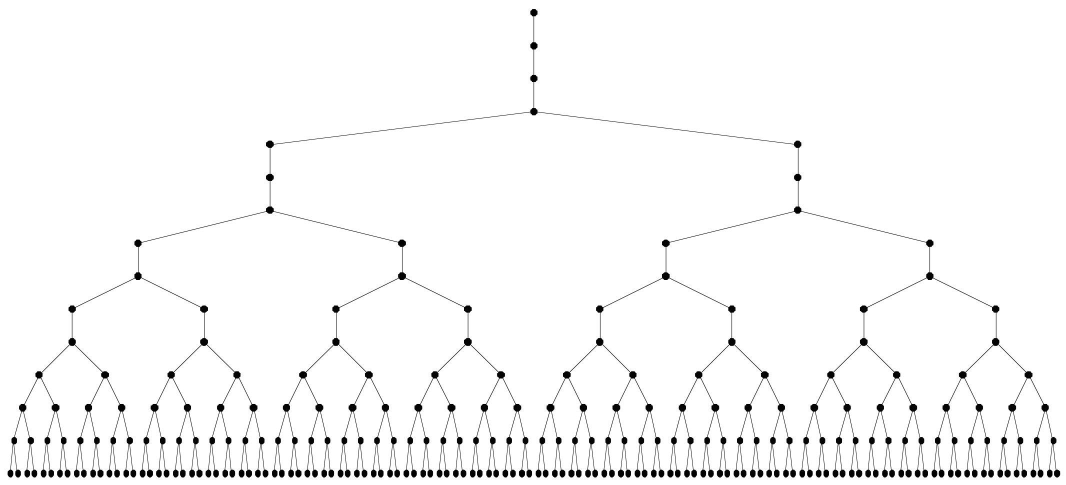
Obviously, a planner will not necessarily examine every node in its serialized search tree. It may prune some of these nodes as infeasible, and it may find its desired solution before it examines all of the unpruned nodes. However, in the worst case, the planner will need to examine every one of the nodes in the serialized search tree. In such a case, a planner that searches the tree S_3 of Figure 3.12 will likely be more efficient than a planner that searches the trees S_1 or S_2.



(a) Basic pattern.



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



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

Figure 3.13: A basic pattern consisting of an AND-branch leading to two OR-branches, an AND/OR tree formed by repeating this pattern, and the smallest possible serialization of the AND/OR tree.

3.2.2 Best and Worst Serializations

If a serialization strategy that would always find the smallest serialization of an AND/OR graph could be found, how much would this help? To get an idea of the answer, take the pattern shown in Figure 3.13a, and use it repeatedly to form an AND/OR tree $G_{b,k}$ of height 2k, as shown in Figure 3.13b. In $G_{b,k}$, the number of occurrences of the pattern is

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

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:

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

$T^-_{b,k}$ 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 3.13c, 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 $T^+_{b,k}$.

We can calculate the node counts for $T^-_{b,k}$ by tracing at which levels b-ary OR-branches branches appear. The total number of nodes in $T^-_{b,k}$ is

$$\begin{aligned} n(T^-_{b,k}) &= 1 + k + bk + b^2(1+b)(k-1) + \cdots + b^{2^{k-1}}\frac{b^{2^{k-1}}-1}{b-1} \\ &= 1 + k + \sum_{i=0}^{k-1}\{b^{2^i}[\sum_{j=0}^{2^i-1} b^k](k-1)\} \\ &= 1 + k + \sum_{i=0}^{k-1}\{b^{2^i}\frac{b^{2^i}-1}{b-1}(k-i)\} = \Theta(b^{2^k}) \end{aligned}$$

Similarly, we can calculate the node counts for $T^+_{b,k}$ by tracing at which levels unary OR-branches branches appear. The total number of nodes in $T^+_{b,k}$ is

$$\begin{aligned} n(T^+_{b,k}) &= \frac{b^{k+1}-1}{b-1} + b^k\frac{b^k-1}{b-1} + b^{2k-1}(1+b^{k-2})\frac{b^{k-1}-1}{b-1} + \cdots + 2^{k-1}b^{2^{k-1}} \\ &= \frac{b^{k+1}-1}{b-1} + \sum_{i=0}^{k-1}\{b^{(k-i+1)2^i-1}[\sum_{j=0}^{2^i-1} b^{j(k-i-1)}]\frac{b^{k-i}-1}{b-1}\} \\ &= \frac{b^{k+1}-1}{b-1} + \sum_{i=0}^{k-1}\{b^{(k-i+1)2^i-1}\frac{(b^{2^i(k-i-1)}-1)}{b^{k-i-1}-1}\frac{(b^{k-i}-1)}{b-1}\} = \Theta(2^k b^{2^k}) \end{aligned}$$

Thus, the numbers of nodes in the worst possible serialization and the best possible serialization differ by a multiplicative factor of $\Theta(2^k)$.

3.2.3 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 3.12, the first choice is whether to expand the OR-branch rooted at `b` or the OR-branch rooted at `f`.

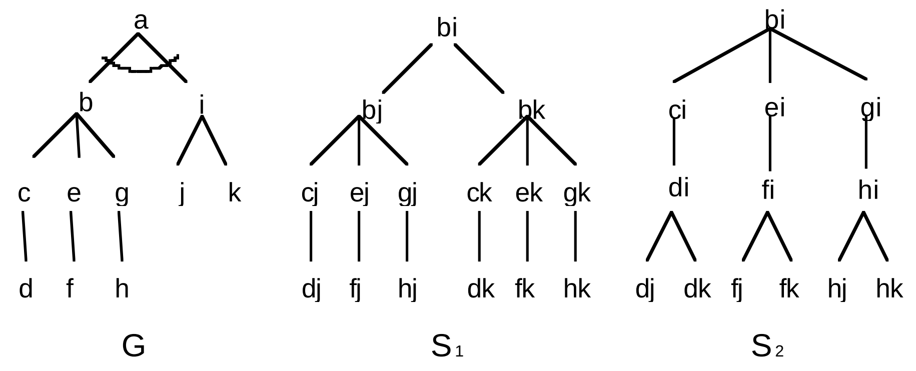


Figure 3.14: A situation where the FAF heuristic fails to produce the best serialization of an AND/OR graph. FAF chooses to expand i before b, thus producing S1; but S2 contains fewer nodes.

If we choose b then we will end up producing a state space similar to S_1 or S_2; and if we choose f then we will end up producing a state space similar to S_3. One way to choose which OR-branch to expand next is to use the fewest alternatives first (FAF) heuristic. In many cases, this simple heuristic produces optimal results. For example, in Figure 3.12, this heuristic would choose to expand f, h, and j before expanding b, thereby producing the tree S_3.

FAF is also 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 bound by some constant b, then the cost of computing FAF is $O(b + b^2) = O(1)$.

Despite its good empirical performance, the FAF heuristic does not always produce optimal results. For example, consider the graph G shown in Figure 3.14. To serialize G, the FAF heuristic 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 counter-example is reminiscent of what happens in a number of NP-hard optimization problems, in which the obvious hill-climbing heuristics for the problems 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 [19].

To formalize the notion of "optimal results" in the previous paragraph, first we defined 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 there is an AND/OR graph G whose root branch is an AND branch. Let the children of the root node be n_1, n_2, $\cdots$,n_k. Then for i = 1,$\cdots$,k, the node n_i is an optimal candidate for expansion if there is a minimal serialization of T of G whose root branch is formed by expanding n_i. For example, f is the optimal candidate in Figure 3.12, and b is the optimal candidate in Figure 3.14. Finding an optimal candidate for expansion is probably NP-hard.

3.2.4 Experimental Studies of FAF

As the experimental results in the previous section and the other studies [20] show, FAF performs well in many domains, but there exist cases where it does poorly. This raises two important questions. First, although FAF performs well in comparison with other popular heuristics, it is not known how close it comes (on the average) to finding the best possible serialization. Second, it is useful to know how it compares, on the average, with the best, worst, and/or average serializations. To try to answer these questions, this subsection presents the results of an experimental exploration.

The performance of FAF was compared 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 [29]. 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 number of nodes 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, a program is used 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, this program was run for up to 50,000 serializations. If the first program had not enumerated all serializations by this cut off (i.e. there were more 50,000 serializations for the given tree), a separate algorithm is used which randomly generated 50,000 trials instead. 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 3.1.

Tree	No. of possible serializations	Size of smallest serializations	Size of largest serializations	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

Tree	No. of possible serializations	Size of smallest serializations	Size of largest serializations	Average serialization size	Size of serialization found by FAF
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

Table 3.1: Experimental results.

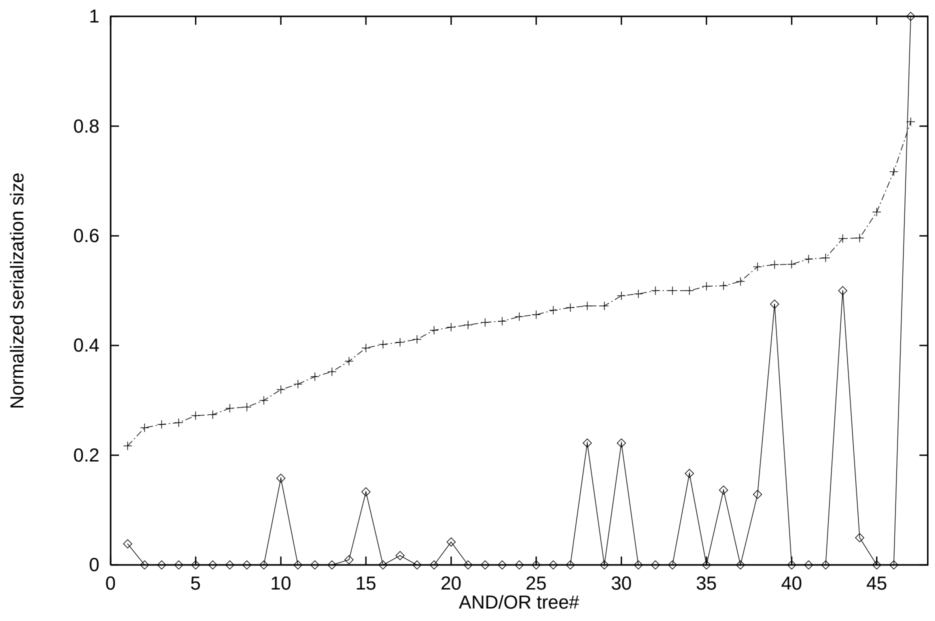


Figure 3.15: Sizes of FAF($\diamond$) and average($+$) serializations normalized with the smallest and largest serialization sizes.

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, a means to measure performance is necessary. In this case, the normalized results were used, with 0 representing the best overall serialization and 1 representing the worst. Figure 3.15 shows the performance of FAF vs. the average.

In 46 of the 47 cases, FAF performs better than the average. In 32 cases, the optimal is found, and in 44 cases, the algorithm performs better than half way between optimal and average. They seem to be quite encouraging results, showing that the FAF algorithm performs quite well in the average case.

There was, however, one case in which FAF produced the worst serialization

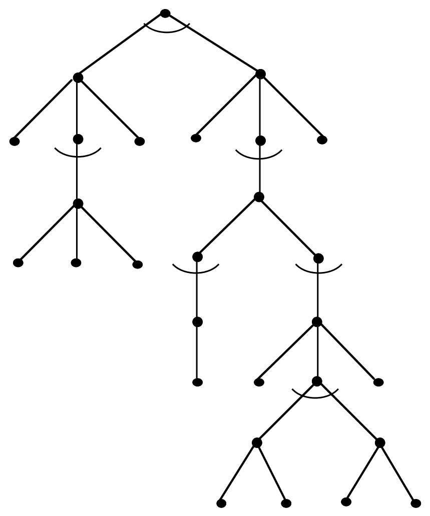


Figure 3.16: The one tree in which FAF produced the worst serialization.

(and, in fact, this is also the only case where FAF did worse than average). The cause of this can be analyzed by looking at the particular tree (shown in Figure 3.16). In this case, FAF could have produced the best serialization if it chose the right child of the root to expand first instead of the left child. Since the program used simply chose leftmost in the case of the tie, FAF did poorly in the test. This does, however, show a potential weakness in implementations of FAF for planning, since it needs to have an additional heuristic to use in these cases. An examination of what to do in this case could lead to further improvement of planning choice mechanisms.

3.2.5 Summary

The search process that is carried out by an AI planning system is shown to correspond to 'serializing' an AND/OR graph-mapping it into an equivalent state-space graph. Different refinement strategies for planning thus correspond to different strategies for serializing the AND/OR graph representing the planning choice points. Different serialization strategies produce state spaces of different sizes, and the smallest serialization of an AND/OR graph can be exponentially smaller than the largest one. A planner whose serialization strategy produces a small state space is likely to be more efficient than a planner whose serialization strategy produces a large state space. The above studies has shown that choosing an efficient strategy can save exponential time.

Like most greedy heuristics, the FAF strategy does not always produce the smallest possible serialization—but in the experimental results, it usually produced a serialization that was either optimal or near-optimal. The studies presented above suggest that the FAF strategy provides a good balance between the complexity of computing the heuristic and the size of the resulting state space.

These results explain why FAF performs well in the previous studies, and opens several interesting issues for exploration. First, as was noted, better serializations lead to smaller search spaces, thus potentially improving planning behavior. However, the exact relationship between a given planner and this search space is quite complex to describe, 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. Overall, the FAF heuristic seems to be a good selection method when there is no good reason to prefer one refinement over others.

3.3 Related Work

As mentioned before, the FAF heuristic was used in constraint satisfaction problems as early as 70s. Bitner and Reingold [6] provided the 'search rearrangement' method as one way to increase the speed of the search: Instead of setting the variables in a fixed order, the search is rearranged by choosing a variable that offers the fewest alternatives. Purdom [40] did statistical analyses on the search rearrangement method applied to SAT problems. By varying the number of clauses and the probability that a literal appears in the clause, he found a class of problems where the search rearrangement method can probably save exponential time compared with the ordinary backtracking method.

The O-Plan planner [10] uses the branch-1/branch-N heuristic as one of the assessment measures to decide which planning operation should be done during the planning process. Branch-1 is a version of FAF which gives the number of immediate search branches the operation generates. Branch-N gives an estimate of the number of distinct alternatives that might be generated by working on the operation. Operations with lower branch-1/branch-N estimates take high priority when O-Plan is choosing an operation.

In action-based planning, Joslin and Pollack studied their least-cost flaw repair (LCFR) strategy, a version of FAF, which was introduced in [20] as a good strategy to determine which refinement operations to do next in a causal-link based planner. In the following studies, Pollack, et al. [38] investigated the performances of LCFR compared with many other strategies. From the experiments they performed, they found that, except for the Tileworld domain problems, LCFR was generally outperformed by a modified LCFR strategy that always delays threat removal operations when the threats are resolved by separation. They reasoned that this is because those threats are often partially resolved by other refinement operations if they are postponed. In the Tileworld domain, the planner has to plan for moving from one location to the other in order to pick up tiles and fill holes with them. Since a plan contains many instances of the operator (GO X, Y) to move around, it is important to put orderings between these actions by resolving threats. Without it, the planner has to handle partial plans with many GO actions without knowing which location each GO originates from. Thus, doing those threat removal operations early is essential to the planner's efficiency. The modified LCFR strategy, which delays those threat removal operations when they are separatable, also delays the pruning, while the LCFR strategy resolves the threats earlier and performs better.

Chapter 4

External Conditions Method

In the previous chapter, the FAF heuristic was shown to do well in minimizing the size of the search tree when it was applied to select alternative refinements in the refinement search. These analyses suggested that if the FAF heuristic was used throughout the planning process, it should perform well for many problems. In reality, it did not. The reason is that the outcome of a refinement operation does not always remain the same during the planning process, because many plan refinement operations are inter-dependent. A refinement operation r which results in N partial plans if it is applied to the partial plan P may result in M partial plans where N $\neq$ M if it is applied to a partial plan P' which is a consequence of another refinement s applied to P. This motivated us to look at interactions that occur during the planning process.

This chapter presents a refinement strategy that reduces the cost of the backtracking that results from a certain kind of task interaction. The strategy, called ExCon, performs better than FAF when used to select which task to decompose next. The first section looks carefully at why FAF does not always work well, and what a refinement strategy should do to perform better than FAF. We analyze how tasks interact in HTN planning in Section 4.2. Section 4.3 defines a type of constraint called an external condition that an HTN planner can use to detect possible task interactions during the planning process. Section 4.4 presents the ExCon strategy that uses external conditions in order to handle task interactions more efficiently. We implemented the ExCon strategy in the UMCP planner. The implementation details are in Section 4.5. Section 4.6 compares FAF and ExCon empirically. Related work is presented in Section 4.7. Finally, Section 4.8 summarizes the chapter.

4.1 Why FAF does not always perform well

As presented in Section 3.3, the empirical study by Pollack et al. [38] in action-based planning shows that a modified FAF strategy which delays certain refinements does better than their default FAF strategy. This is because a delayed refinement is often resolved as a result of doing other refinements. In other words, other refinements applied during the interval it was delayed often prune the search space in such a way that the delayed refinement will not partition the refinement search space. Similarly, refining a partial plan in HTN planning often fully or partially resolves the other refinement operations. Also, doing several refinements in combination may help the planner prune the search space more efficiently. Consider the following examples:

- Suppose there are two refinements r_1 and r_2, among many others, that are applicable to the current partial plan P (see Figure 4.1(a)). r_1 enforces a constraint c_1 that can be satisfied by binding a variable ?x with a constant a. r_2 enforces another constraint c_2 that can be satisfied by binding the variable ?x with a, b, or c (i.e. by setting the possible values for ?x with {a, b, c}). If

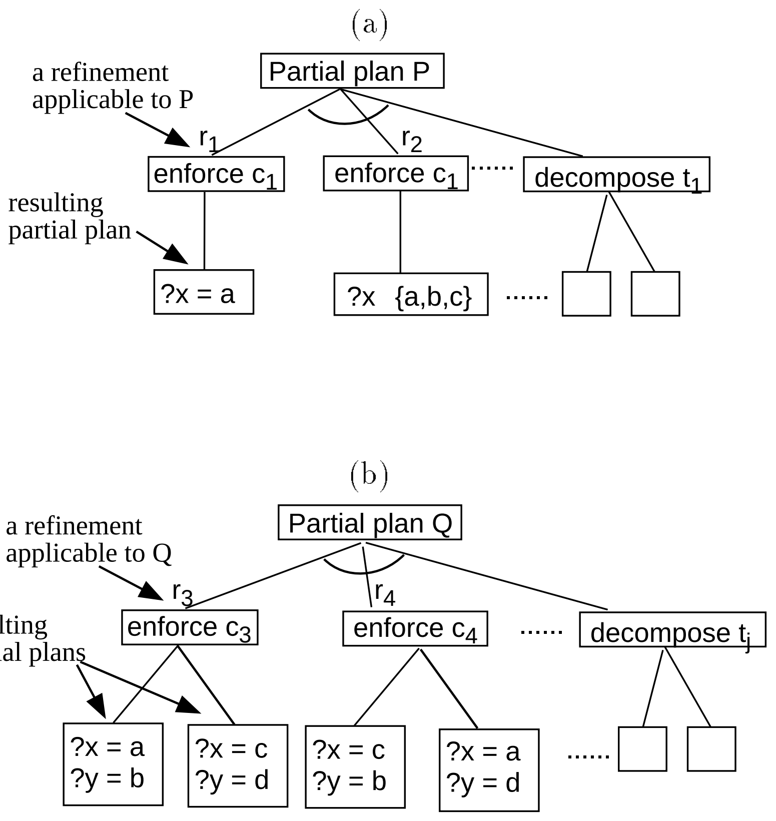


Figure 4.1: The outcome of a refinement may change if it is delayed. In (a), the planner does not need to do r_2 if r_1 was applied first. In (b), doing r_1 and r_2 will prune the search space represented by the partial plan Q. However, if the planner works on refinements other than r_1 or r_2, it may have to spend a lot of time backtracking for those refinements.

the planner's refinement strategy chooses to do r_1 first, then the constraint c_2 will be satisfied and the planner no longer needs to work on enforcing the constraint c_2. On the other hand, if the refinement strategy chooses to do r_2, the planner still needs to enforce c_1 later. Since each of r_1 and r_2 generates one immediate partial plan, FAF may do the former or latter.

- Suppose there are two refinements r_3 and r_4, among many others, that are applicable to the current partial plan Q (see Figure 4.1(b)). r_3 enforces a constraint c_3 that can be satisfied by binding variables so that either ?x = a and ?y = b, or ?x = c and ?y = d. r_2 enforces a constraint c_4 that can be satisfied by binding variables so that either ?x = c and ?y = b, or ?x = a and ?y = d. Thus, doing either r_1 or r_1 generates two immediate partial plans. However, applying the two refinements results in pruning any partial plans that are derived from Q, because of the inconsistency. If there are other refinements that generate less than two immediate partial plans, FAF will choose to do those refinements before choosing r_1 or r_2 although doing so will later result in bigger backtracking costs.

As shown by the above examples, the number of partial plans generated by a refinement may not remain the same if the refinement is postponed. Since the FAF heuristic takes into account only the number of immediate partial plans that result from each refinement, it may not perform as well as strategies that look at the interactions between refinements operations. This is not to say that FAF should not be used at all. It is very difficult for any planner to know the outcome of a refinement if it is postponed, since that depends on what other refinements are performed in the meanwhile. The FAF heuristic has a good chance of reducing the size of the search space when the planner does not have a particular reason

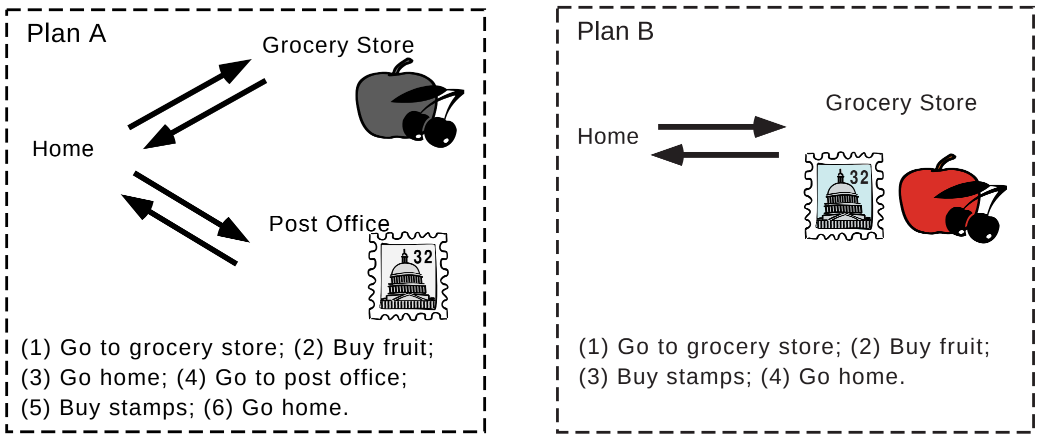

Plan A
Grocery Store
Home
Post Office
32
(1) Go to grocery store; (2) Buy fruit;
(3) Go home; (4) Go to post office;
(5) Buy stamps; (6) Go home.
Plan B
Grocery Store
Home
32
(1) Go to grocery store; (2) Buy fruit;
(3) Buy stamps; (4) Go home.

Figure 4.2: Two possible plans to buy fruit and stamps.

to expect other refinements to do better. However, this motivated us to look at possible interactions that occur during the planning process in HTN planning. Specifically. we investigated interactions between task decompositions since our preliminary study showed that much of the backtracking costs result from failures to resolve task interactions efficiently.

4.2 Task Interactions

Gupta and Nau [19] presented two types of task interactions that pose efficiency problems. A *deleted-condition interaction* is a situation where one action, which is inserted into a plan to achieve one goal, deletes a condition necessary to accomplish another goal. This type of interaction has already been addressed in many literatures [41, 14, 26]. The common approach to handle it is to detect possible threats (i.e. actions that can delete necessary conditions), and remove threats by constraining variable bindings or step orderings. If a threat cannot be removed, the planner backtracks. A planner can reduce such backtracking costs by detecting and handling threats as soon as possible.

An *enabling-condition interaction* is a completely different kind of interaction. It occurs when an action that is introduced into a plan to accomplish one goal makes it easier to achieve another goal. Unlike a deleted-condition interaction, it is, in a way, beneficial because it is usually desirable to do one action for more than two purposes, rather than do actions for each purpose separately. For example, consider a situation where you want to get fruit and stamps and be back at home. [1] One plan to accomplish it is to go to a post office, buy stamps, go home, leave for a grocery store, buy fruit and come back (as shown in Plan A of Figure 4.2). But if you know a grocery store which sells stamps, then you probably would rather go to the store and buy both fruit and stamps there, and then come back as shown in Plan B of Figure 4.2. In other words, Plan A shows a plan where the task of buying fruit and the task of buying stamps are not interleaved, and Plan B shows a plan where they are. In this thesis, "tasks X and Y are interleaved" means that the planner has put orderings between X and Y in order to utilize enabling-condition interactions, even if it resulted in a sequential ordering between X and Y (i.e. every action for X comes before any action for Y, or vice versa).

In their analysis of complexities, Gupta and Nau [19] show that finding an optimal solution in Blocks World is NP-hard. Moreover, they show that the NP-hardness is due to enabling-condition interactions, not to deleted-condition interactions. This result motivated us to look at enabling-condition interactions in HTN planning more carefully.

Figure 4.3 shows two alternative decomposition methods for the Get-Stamps task. Method 1 specifies buying stamps in a store which sells stamps. One usually does not go to a grocery store solely to buy stamps. Method 1 is applicable only

[1]This example is based on Wilensky [52].

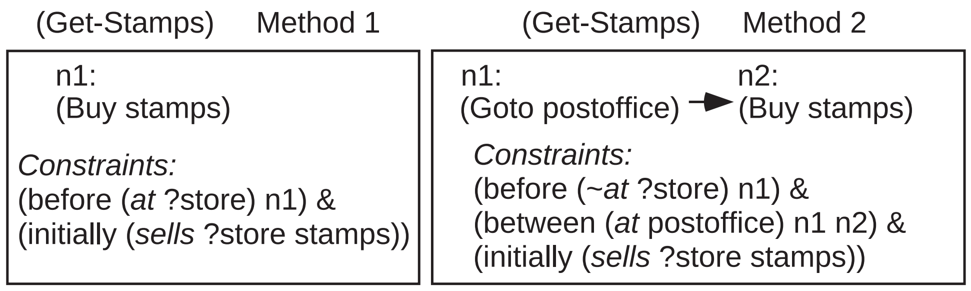

(Get-Stamps) Method 1
n1:
(Buy stamps)
Constraints:
(before (*at* ?store) n1) &
(initially (*sells* ?store stamps))
(Get-Stamps) Method 2
n1:
(Goto postoffice)
n2:
(Buy stamps)
Constraints:
(before (~*at* ?store) n1) &
(between (*at* postoffice) n1 n2) &
(initially (*sells* ?store stamps))

Figure 4.3: Two methods to get stamps

when one is at the store for some other reason and has the condition (*at* ?store) as a 'before' state constraint instead of as a predicate task. Method 2 specifies a procedure of buying stamps in a post office when one is not at a store which sells stamps. Thus, decomposing a Get-Stamps task creates two partial plans; one using method 1 and one using method 2. Suppose the planner explores the one using method 1 in the next step. In order to decide if the method used is applicable, the planner needs to examine the initial state and actions that come before the (Buy stamps) task to see that the condition (*at* ?store) can be satisfied. However, if there are non-primitive tasks that could come before the (Buy stamps) task, a planner may not be able to decide whether the condition can be satisfied or whether it is unsatisfiable at that time. Of course, the planner will eventually work on (i.e decompose) each one of these non-primitive tasks and thus be able to tell if the condition is satisfiable or not. But the cost of backtracking can be high if the planner finds the condition unsatisfiable (i.e. the method inapplicable) long after it is inserted into the plan.

4.3 External Conditions

We can separate state constraints into three types based on what establishes them. One type of constraint is a constraint on the state conditions that never change throughout the plan. These constraints can only be established by the initial state. Conditions of this type are called *initial state constraints.* An initial state constraint can only be established by binding variables in the constraint to match what is true in the initial state. The second type of constraint is a constraint that can be established within the subtasks of the decomposition method. The third type of constraint is a constraint that must be established externally, either by the initial state or a task. We call constraints of this type external conditions.

4.3.1 Definition

An external condition of a decomposition method M is a state constraint that has to be established for the method M to be applicable to the problem and yet cannot be established by any task that may result from M. Thus, the plan must establish this state constraint by something external to M, such as the initial state or some other task in the plan.

Definition (External condition) Let $M =< T, \phi >$ be a decomposition method, where T is the set of subtasks created by the method and ϕ is the set of constraints for the method. Then a condition c is an *external condition* of M if:

1. c is a state constraint other than an initial state constraint;
2. c must be necessarily true to satisfy ϕ; and
3. no descendent of any task in T can establish c.

In the definition, external conditions are constraints that definitely have to be established by something external to the method. Since what the tasks in T are decomposed into depends on the context of each problem, there can be a state constraint that can be established by a descendent of T in some problems and yet cannot be established by a descendent in other problems. Such a constraint will not be considered as an external conditions of the method.

Notice that an external condition of a method M is not external to the task M is decomposing; when a non-primitive task has more than one decomposition method, each method may have different set of external conditions.

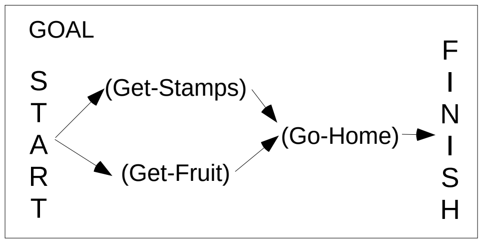
As an example, consider the external conditions of each method for the Get-Stamps tasks shown above. For method 1, there are two conditions. The constraints, (before (*at* ?store) n1) and (initially (*sells* ?store stamps)), are state constraints that must be necessarily true to satisfy the constraint formula and cannot be satisfied by a descendent of any task in the method. However (initially (*sells* ?store stamps)) is an initial state constraint which is never affected by other tasks. So, (before (*at* ?store) n1) is the only external condition for method 1. For method 2, there are three conditions and all of them must be true to satisfy the constraint formula. The constraint (before (∼*at* ?store) n1) is an external condition of the method as no descendent of any task in the method possibly can establish it. On the other hand, (between (*at* postoffice) n1 n2) is not an external condition because the task n1:(Goto postoffice) can make the condition (*at* postoffice) true. (initially (*sells* ?store stamps)) is not an external condition since it is an initial state constraint.

4.4 ExCon Strategy

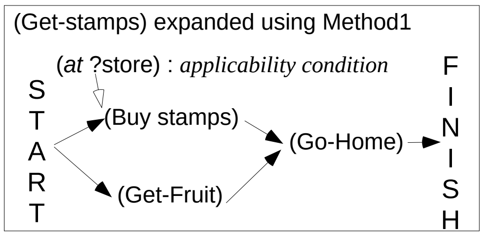
When a decomposition method is instantiated, the external conditions in the methods become *applicability conditions* of the plan, i.e. conditions that must be established for the partial plan to work for the current problem. Since these conditions cannot be established by any task introduced by the method, they must be established by either another task or the initial state. If there are non-primitive tasks that can come before the point where the condition must be true, the planner may not be able to fully establish the applicability condition at the current time. However, postponing the establishment of applicability conditions may lead to huge backtracking costs when some condition turns out not to be establishable. In order to reduce such backtracking costs, the planner needs to work first on tasks that *might* effect the establishment of the applicability conditions of the partial plan it is working on. This section describes the strategy that selects which tasks to decompose next based on the current applicability conditions of the partial plan.

4.4.1 Illustration

Consider the situation presented in in Section 2.3 where you have to (1) get stamps, (2) get fruit, and (3) then go home. This goal can be represented as a partial plan as shown in Figure 4.4 (a). If a planner chooses to work on the task (Get-Stamps) first, decomposing (Get-Stamps) in this partial plan will result in two partial plans, one using Method 1 (as shown in Figure 4.4 (b)) and another using Method 2 (not shown). Now, consider the one using Method 1. Suppose the task (Get-Fruit) can be decomposed into a sub-plan that includes going to a store which sells stamps, then the condition (*at* ?store) can be satisfied at the beginning of the task (Buy stamps). However, the planner cannot know if such a sub-plan for the task (Get-



(a) A partial plan specifying the goal of getting stamps, fruit and gas and going home.


(Get-stamps) expanded using Method1
(*at* ?store) : *applicability condition*
START
(Buy stamps)
(Get-Fruit)
(Go-Home)
FINISH

(b) A partial plan resulting from the (Get-Stamps) task expanded using method 1.

Figure 4.4: A sample applicability condition in a partial plan

Fruit) is applicable to the current problem until after it has worked on (i.e. fully decomposed) the task.

The planner has to choose a non-primitive task such as (Get-Fruit) or (Go-Home) in the partial plan to work on next. If sometime later the planner finds that in the current problem, accomplishing the task (Get-Fruit) does not include an action of going to a store which sells stamps, then the entire search derived from the partial plan fails and the planner has to backtrack. The cost of such backtracking can be significantly high if the planner worked on the task (Go-Home) down to the detailed level and then found the failure while working on the task (Get-Fruit).

The idea of the ExCon strategy is to work first on the tasks which can help the planner establish the applicability conditions, thus reducing the cost of backtracking when the planner finds conditions that cannot be established.

4.4.2 Method

The ExCon strategy requires the following:

1. When it loads its knowledge base containing the domain specification, the planner must precompute external conditions for every decomposition method in the domain and store the information. The planner only needs to do this once for a domain. While the complexity of extracting every external condition is probably undecidable, we can extract most external conditions in polynomial time using a simple algorithm. We describe this algorithm later when describing the implementation of the ExCon strategy.

2. The data structure for a partial plan keeps a stack of applicability conditions. Initially, the partial plan has no applicability conditions. During a method instantiation, the external conditions of the method are pushed onto the top of this stack. The condition on top is the current priority to the planner.

3. When selecting a task to decompose, the priorities are given to (1) tasks which can possibly establish the current top condition, or (2) tasks which can possibly threaten the current top condition, based on the presence of a primitive establisher.

The algorithm for the third step (selecting a task to decompose) is shown in Figure 4.5. For selecting tasks in Steps 1, 4, and 5, the algorithm uses whatever task-selection strategy the user wishes (FAF is used for this purpose in the experiments described in this section).

In Step 1, if there are no applicability conditions to achieve, then the planner selects and returns a task. When there are applicability conditions, Step 2 picks up the one on top of the stack. If the current condition is already established without threats in the partial plan, then Step 3 removes the condition from the stack and goes back to select something else. Otherwise, Step 4 computes the non-primitive

```
Algorithm select-task-ExCon($PartialPlan$)

    1. If the applicability condition stack of $PartialPlan$ is empty, then select a
       task from $PartialPlan$ and return the result.

    2. Else, set $c$ to the first element of the applicability condition stack in
       $Partial\ Plan$.

    3. If $c$ is true in $PartialPlan$, then remove $c$ from the applicability condition
       stack and go back to Step1.

    4. If there is no primitive task that establishes $c$, then compute possible
       establishers for $c$. Select a task among them and return the result.

    5. Else, compute possible threats for $c$. Select a task among them and
       return the result.

    6. If there are no possible establishers or possible threats, remove $c$ from
       the applicability condition stack of $PartialPlan$ and go back to Step 1.
```

Figure 4.5: The task selection algorithm for ExCon.

tasks in the plan that can possibly establish the condition, provided the condition is not established by any primitive task currently in the plan. If it is established by a primitive task, then possible threats are computed and one is selected in Step 5. Otherwise, there are only primitive tasks that might affect this condition, so Step 6 will remove it from the stack and go back to select another one.

Note that since ExCon's task-selection strategy merely specifies the order in which a planner will prefer to expand tasks, it has no effect on the planner's completeness: a planner that is sound and complete without it will also be sound and complete with it.

4.5 Implementation

We implemented FAF[2] and ExCon using the UMCP planning system, a domain-independent HTN planning system [11], and compared their performances. For the experiments, we ran UMCP version 1.2[3] on Sun ULTRA workstations using Allegro Common Lisp 4.3. We incorporated each task selection strategy into UMCP's default commitment strategy, which is described in Section 2.3.3. The domain descriptions used in the experiments are available at
⟨http://www.cs.umd.edu/projects/plus/umcp/domains/⟩.

[2]In the implementation of FAF, the FP heuristic as the tie-breaking rule for FAF. The FP heuristic is describe in Section 5.

[3]Note: The previous experiments on ExCon used UMCP1.0. UMCP1.2 is capable of pruning more plans than UMCP1.0 by looking at possible effects in more detailed level.

4.5.1 Automatically extracting external conditions

Computing precisely which state constraints are external conditions is not a trivial matter since it requires the planner to know the exact variable bindings that can occur during the planning. To see which tasks affect which constraints, UMCP1.2 uses a *possible effects table* to store information about which non-primitive tasks are capable of causing various kinds of effects. Since the exact effect of each non-primitive task depends on which decomposition methods are used and how the variables bound, the table only specifies which non-primitive task can possibly affect each predicate. The table is a table of pairs $< p, t >$ which p is a positive or negative atom and t is a non-primitive task. An argument of p can be bound to some argument of t only if the effects of t on the predicate is limited to it. Otherwise, the arguments will be indicated as "??". For example, if a task (move ?robot ?loc) has a possible effects of (*at* ?robot ??), then the task (move Robot1 RoomA) may cause the effect (*at* Robot1 Hallway), but never cause the effect (*at* Box RoomA).

A possible effects table is computed by exploring all possible decompositions of each non-primitive task. A pair $< p, t >$ is in the table if one of the possible decompositions of the task t contains a primitive task s such that one of s's effects is the literal p.

The possible effects table can be constructed by a planner by preprocessing the domain. During the planning, the planner can look in the table to see which non-primitive task in the current partial plan can possibly establish or threaten certain constraints in order to prune partial plans that have no way of satisfying necessary state constraints.

In order to extract the external conditions of each method in the domain, the planner needs to do the following for any method $M = < T, \phi >$ in the domain:

- First, all the non-initial state constraints in ϕ are pushed onto a set C. If a constraint is specified in a disjunctive form (i.e. $c_1 \vee c_2 \vee \cdots$) in ϕ, it is removed from C.

- Then, for every constraint c in C, do the following:

  – Determine at which state s, the condition in c must be satisfied. For example, for the condition (before (*at* ?store) n1), the constraint (*at* ?store) must be satisfied at the beginning of the task n1. Similarly, for the constraint (between (*at* postoffice) n1 n2), the condition must be satisfied in the state immediately following the task n1.

  – Look for a task $t_i \in T$ where (1) there are no ordering constraints in ϕ that order t_i after s, and (2) there is a pair $< p, t_i >$ in the possible effects table where the literal p can be unified with the condition in c.

  – If such t_i cannot be found, c is marked as an external condition to M.

This algorithm takes only polynomial time (to the size of the domain). It will not necessarily extract all external conditions, since the variable bindings are not fully examined. We are currently trying to find a better way to extract them. However, in all of our test domains, the above algorithm extracted most of the external conditions. Furthermore, as will be shown below, this set is enough to significantly improve planning behavior.

4.5.2 Computing possible establishers and possible threats

As shown in Figure 4.5, Steps 4 and 5 of ExCon's task-selection compute the possible establishers and the possible threats of the applicability condition. In our implementation, the planner uses the possible effects table to compute these. First, the planner finds all non-primitive tasks in the partial plan that are not ordered after the point where the condition needs to be true. It then looks in the possible effects table to see if any of them can possibly establish or threaten the condition, and returns the result. Although this method returns (as possible establishers or possible threats) some tasks that can never establish nor threaten the conditions, it finds every possible establisher and possible threat to the condition.

4.5.3 Plan selection

Since Blocks-World has recursive tasks where the search space can be infinite, our experiments used best-first search for Block-World problems and depth-first search for all the other problems.[4] For best-first search, plan selection is based on the value computed by,

$$\begin{aligned} \mathrm{f}(\mathit{PartialPlan}) = {} & \text{(number of non-primitive tasks)} \\ & + \text{(number of tasks, both primitive and non-primitive)} \\ & + \text{(number of ordering and variable constraints that} \\ & \quad \text{are postponed).} \end{aligned}$$

The plan with the lowest value is selected for next refinement. This function was created based on the heuristic presented by Gerevini and Schubert [18] and seems to perform well on many problems with infinite search space.

[4]We used depth-first search for finite search space problems because (1) it is easier to trace the search and (2) it generally does as well as or better than best-first search on our test domains.

4.6 Experiments

Since both the FAF and ExCon strategies merely specify the order in which a planner will prefer to decompose tasks, they have no effect on the planner's soundness and completeness. However, they do affect the planner's efficiency; and the ExCon strategy should outperform the FAF strategy, especially on problems where the goal tasks are highly interleaved.

We tested ExCon against FAF on three domains. One is an artificial domain where the amount of interleaving can be controlled to some degree. The second domain is UM Translog. We tested the two strategies on various types of one-, two-, and three-package problems. The third domain is Blocks World. We used tower inversion problems with different number of blocks. We measured the number of partial plans the planner created before finding a solution plan.

4.6.1 Artificial Domain

The test domain contains methods for accomplishing compound tasks called p-task, q-task, and r-task. As shown in Figure4.6, these methods decompose the compound tasks into other tasks. Most of the other tasks are primitive tasks, but a few of them (p, q, and r) are predicate tasks. Most of the primitive tasks (do-p1, do-p2, do-q1 do-q2, do-r1and do-r2) have no preconditions and effects. Each predicate task has two methods that are capable of achieving it: one of the methods shown in Figure 4.6(d)–(f), and a Do-Nothing method.

The primitive task (del-p ?x ?y) has the effects (∼*p* ?y) and (*prep* p). The task (set-p ?x) has the effects (*p* ?x) and (∼*prep* p). The predicate task (p W) for some value W can be achieved in two ways; by phantomizing it if the literal (*p* W) is true at the beginning of the task (p W); or by doing (del-p W Z) followed by (set-p W) if at

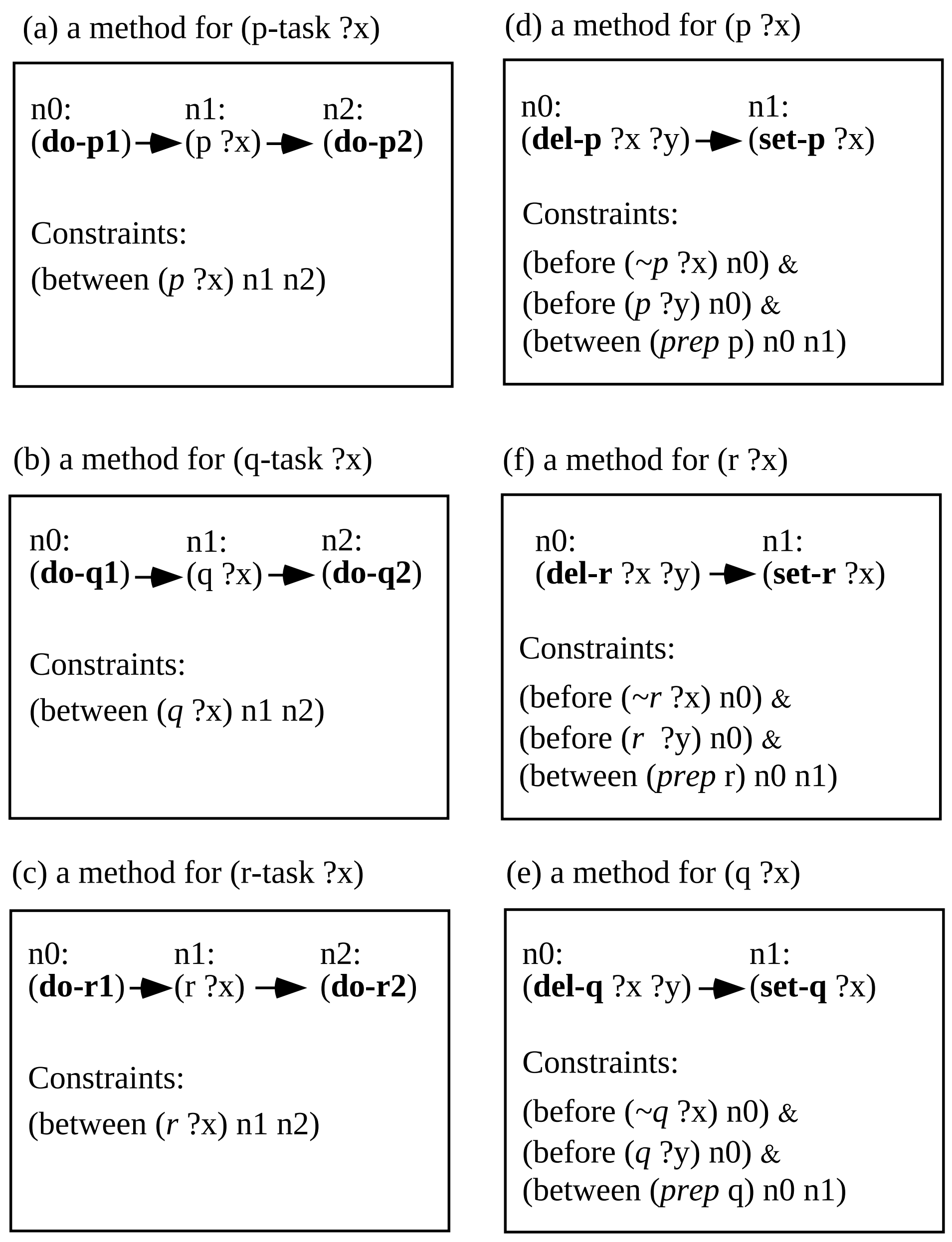


Figure 4.6: The decomposition methods for the test domain. Each non-primitive task has exactly one method specified. The tasks shown in boldface are primitive tasks.

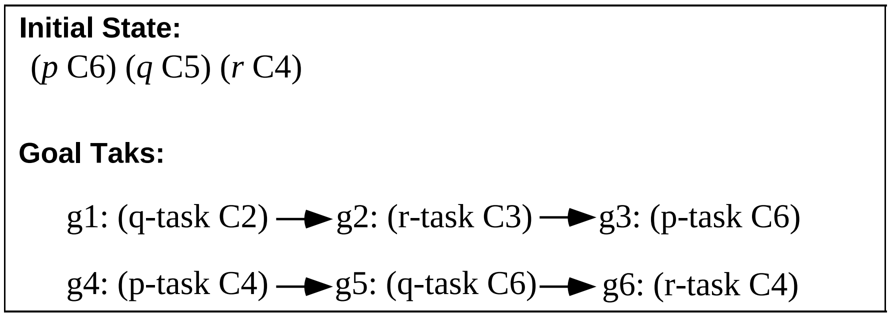

Initial State:
(p C6) (q C5) (r C4)
Goal Taks:
g1: (q-task C2) → g2: (r-task C3) → g3: (p-task C6)
g4: (p-task C4) → g5: (q-task C6) → g6: (r-task C4)

Figure 4.7: A sample problem with 2 goals, 3 predicates and 50% overlap. We represents the initial state which consists of (*p* C6), (*q* C5) and (*r* C4).

the beginning of the task (p W) the literal (*p* W) is false and the literal (*p* Z) is true for some value Z. The tasks del-q and del-r are defined similarly to the task del-p, and the tasks set-q and set-r are defined similarly to the task set-p. An initial state for this domain consists of three ground atoms (*p* w_p), (*q* w_q) and (*r* w_r), where w_p, w_q and w_r are constant values randomly chosen from the set { C1, C2, C3, C4, C5, C6 }.

In this test domain, the amount of interleaving can be altered by varying the arguments of the goal tasks: a problem is highly interleaved if the arguments of most p-task goal tasks are the same and it is less interleaved if the arguments of most p-task goal tasks are different.

We generated test problems as follows. Goals were random sequences of one (p-task only), two (p-task and q-task) or three (p-task, q-task and r-task) different tasks that needed to be done. How many different tasks were in the goal decided the number of predicates in the problems. A problem consisted of two or three goals, with no ordering constraints across them. We randomly assigned arguments to the tasks based on an "overlap rate" of 10%, 50% or 90%. For example, if the overlap rate were 100%, all the arguments of the p-task tasks would be identical

	FAF	ExCon	p	FAF	ExCon	p	FAF	ExCon	p
2 goals	1 predicate			2 predicates			3 predicates		
90%	10.3	10.3	–	22.7	21.5	95.5%	33.3	32.0	>99.9%
50%	11.2	11.2	–	24.5	23.7	>99.9%	39.1	34.7	>99.9%
10%	11.7	11.7	–	24.8	24.6	92.7%	37.4	36.4	>99.9%
3 goals	1 predicate			2 predicates			3 predicates		
90%	16.3	16.0	99.5%	36.0	35.6	67.1%	63.1	49.6	>99.9%
50%	26.3	25.5	98.1%	81.9	48.5	>99.9%	420	78.9	>99.9%
10%	32.6	32.7	66.7%	141	66.9	>99.9%	515	91.0	>99.9%

Table 4.1: The average numbers of search nodes created over the 100 randomly generated problems in the artificial test domain. p represents the confidence levels of paired t-test.

to the *p* value in the initial state. If the overlap rate were 0%, the arguments for p-task and the *p* value in the initial state would all be unique. If the overlap rate were 30%, there would be 30% probability that the argument of a p-task is used in another p-task or the atom *p* in the initial state. We varied the overlap rate to create problems with various degrees of interleaving. We also varied the number of predicates appearing in the problems to change the chances that the planner would try to interleave multiple predicate tasks. A sample problem is shown in Figure4.7.

We created and tested 100 problems of 1, 2 or 3 predicates used, 10%, 50% or 90% overlap, and 2 or 3 goals, totaling 1800 problems tested. We counted the number of partial plans created during planning and computed the average for each type of problem. The results are shown in Table 4.1.

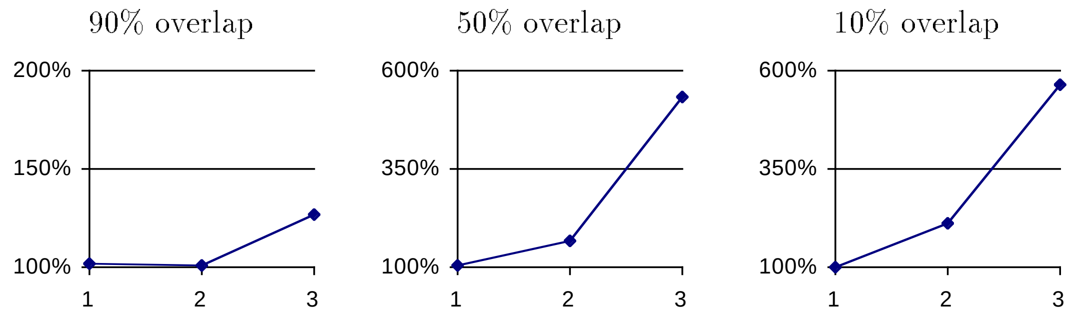


Figure 4.8: The relative performances of FAF and ExCon for 3-goal problems. The x-axis gives the number of predicates in the goals, and the y-axis gives the ratio (#search nodes by FAF)/(#search nodes by ExCon).

Figure 4.8 shows graphs of the 3-goal data in Table 1, that show how the relative performance of FAF and ExCon depends on number of predicates in the goal. Note that as the number of predicates increases, FAF's performance degrades much more quickly than ExCon's. This is because FAF creates many more partial plans in order to interleave goal tasks. Since ExCon works on one predicate at a time until it is established or fails, the planner does not have to backtrack over multiple predicates as it does with FAF.

The graphs in Figure 4.9 show how the relative performances of FAF and ExCon depend on the overlap rate. For problems with 1 predicate, the difference between FAF and ExCon is not large regardless of the overlap rate. For the problems with 2 or 3 predicates in the goals, FAF is clearly spending more time backtracking than ExCon. At 90% overlap, most attempts to interleave tasks succeed. So the amount of backtracking is minimal. Thus, the performances of the two strategies are similar. As the overlap rate decreases to 50% and 10%, less and less attempts to interleave tasks succeed and the planner has to backtrack on many more failures.

Since seeing average numbers alone can be misleading in some cases, we did a

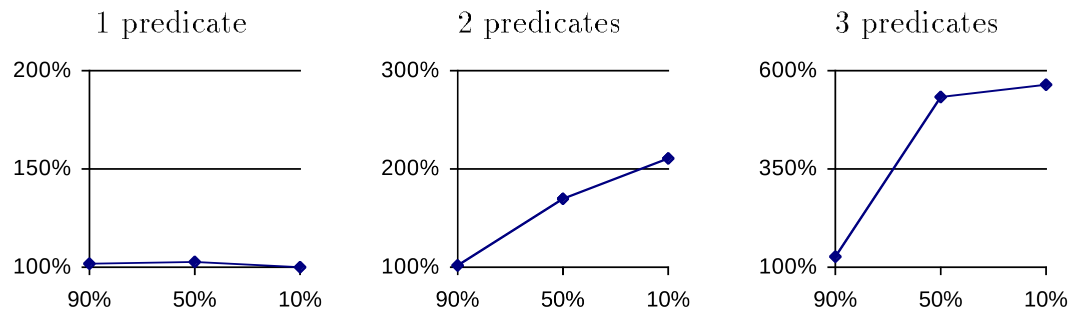


Figure 4.9: The relative performances of FAF and ExCon on 3-goal problems. The x-axis gives the overlap rate, and the y-axis gives the ratio (#search nodes by FAF)/(#search nodes by ExCon). Note: the x-axis must be changed to 90%, 50% and 10%.

one-tailed paired t-test to test the statistical significance of the results. The null hypothesis is $H_0 : \mu_{FAF} = \mu_{ExCon}$ and the counter hypothesis is $H_1 : \mu_{FAF} > \mu_{ExCon}$, where μ_{FAF} and μ_{ExCon} are the means of the numbers of partial plans created by using FAF and ExCon, respectively. The p values in Table 4.1 show the confidence level of rejecting H_0 in favor of taking H_1. If ExCon constantly outperforms FAF, the confidence level is high.

For problems with 2 goals and 1 predicate, there is no difference in the performance of FAF and ExCon, so t-values cannot be computed. For all other problems except 3-goal, 2 predicate, 90% overlap problems and 3-goal 1 predicate, 10% overlap problems, we can reject the hypothesis with a confidence level higher than 90%. On 3-goal, 2 predicate, 90% overlap problems, the planner can successfully interleave many goal tasks without any backtracking for most of the problems. However, for two out of the hundred problems generated, ExCon creates about twice as many partial plans as FAF does to have tasks ordered consistently. (If we do the t-test ignoring these two results, the confidence level is 99.5%.) As for

3-goal, 1 predicate, 10% overlap problems, many problems have goal tasks which cannot be interleaved at all. Since there is only one type of goal task (p-task), FAF can easily identify failures to interleave tasks, thus begins backtracking earlier. So the backtracking costs are small for FAF as well for ExCon. Thus, the t-test results also confirm that the performance of ExCon is significantly better than the performance of FAF when the overlap rate is low and when there are more predicates in the problems. Therefore, ExCon is shown to perform better than FAF on problems where the tasks are highly interleaved.

4.6.2 UM Translog

UM Translog[3] is a transport logistics domain where the methods of transportation are specified based on the locations, the types of the package, and availabilities of the necessary equipments. It is a considerably larger domain than many other toy domains. It is specified with 17 compound tasks, 42 primitive actions, and 29 predicates. We have tested FAF and ExCon on one-, two- and three-package transportation problems. For two- and three-package problems, we differ the possibility of interleaving between different packages by altering initial locations, destinations and package types.

(a) - The package types are of the same type and have the same initial location and destination.

(b) - The package types are of the same type. The destination of one package is the same as the initial location of the other package, so that one truck delivering one package can pick up another package at the place.

Problem	Types	FAF		ExCon		FAF/ExCon	p
		Plans	Time	Plans	Time	(Plans)	
1pack	–	76.00	0.31	76.00	0.35	1.00	–
2pack(a)	same	151.75	1.40	178.20	1.98	0.85	<0.1%
2pack(b)	same	382.30	4.04	229.65	3.04	1.66	>99.9%
2pack(c)	same	1163.20	13.48	536.25	10.37	2.17	>99.9%
2pack(d)	diff	1448.80	16.74	633.40	9.90	2.29	>99.9%
3pack(a)	same	253.60	7.45	238.15	6.28	1.06	>99.9%
3pack(b)	same	81140.65	1,961	33639.15	1,075	2.41	>99.9%
3pack(c)	same	132418.00	3,361	57692.10	1,937	2.30	>99.9%
3pack(d)	diff	44080.45	991	20733.55	556	2.13	>99.9%

Table 4.2: The results for FAF and ExCon on Translog problems. Plans are average number of partial plans created. Time is non-garbage collection CPU time in seconds. For multiple package problems, "Types" shows whether the packages have the same type and so can be carried by the same delivery truck. p represents the confidence levels of paired t test.

(c) - The package types are of the same type but none of the initial locations or the destinations are the same.

(d) - The packages are of different types and so it requires different types of vehicles to transport each package.

We randomly created 20 problems for each problem type, with various package types (regular, bulky, granular, liquid, or livestock) and locations (among 15 locations). The results are shown in Table 4.2.

- For one package problems (1pack), there is almost no interleaving between any tasks in the problem, so the performances of FAF and ExCon are similar. For two-package delivery problems and three-package delivery problems, the performances of the two strategies depends on how much tasks can be successfully interleaved between two goal tasks.

- For the problems where the package are of the same type and have the same initial location and destination (2pack(a) and 3pack(a)), the task of moving trucks to the necessary locations can be completely interleaved between the goals. So FAF can perform as well as ExCon on these problems. In fact, for 2pack(a), FAF outperforms ExCon on the test problems. This is because that the planner instantiates some variables earlier with ExCon, generating many search branches which eventually fail while with FAF, the same variables are instantiated later when the variables are more constrained and thus instantiating them generates less search branches.

- For the problems where the package are of the same type but interleaving tasks between goals work only partially or not at all (2pack(b), 2pack(c), 3pack(b) and 3pack(c)) ExCon outperformed FAF.

- For the problems where the package are of the different types (2pack(d) and 3pack(d)), ExCon outperformed FAF.

We also performed the same one-tailed paired t-test on the Translog results. The results are shown as p in the table. Since the performances of FAF and ExCon are the same for one-package problems, a t-test cannot be done for this problem type. Except for 2pack(a) where ExCon is outperformed by FAF, the performances of FAF and ExCon are significantly different at the 99.9% level. Thus, except for one

Problems	FAF		ExCon		FAF/ExCon
	Plans	Time	Plans	Time	(Plans)
tower-invert3	**45**	**0.23**	52	0.33	0.87
tower-invert4	169	2.73	**138**	**2.17**	1.22
tower-invert5	798	33.15	**521**	**16.65**	1.53
tower-invert6	4921	438.43	**2505**	**184.90**	1.96

Table 4.3: The results on the Blocks World problems. Plans are number of partial plans created. Time is non-garbage collection CPU time in seconds.

package problems (1pack), the performance of ExCon is significantly better than the performance of FAF.

4.6.3 Blocks World

We also ran FAF and ExCon on tower inversion problems in the Blocks World domain. We used 4 problems (tower invert3, tower invert4, tower invert5 and tower invert6) each with a different number of blocks involved. We hypothesized that as the number of blocks increases, ExCon should increasingly outperform FAF because as more blocks are involved, the amount of interleaving increases. The results are shown in Table 4.3. Although FAF outperformed ExCon on tower invert3, ExCon outperformed FAF on problems with more blocks.

As we were testing the strategies, we noticed that the results change if we modify the order in which the goal tasks are specified in the problem. This is understandable for FAF since FAF does not prefer which instance of a task to decompose because every instance has the same number of methods. In such a case, FAF returns the one that happens to be found in the partial plan first. For

Problems	FAF		ExCon		FAF/ExCon
	Plans	Time	Plans	Time	(Plans)
tower-invert3	44	0.54	**32**	**0.22**	1.38
tower-invert4	403	14.66	**70**	**1.12**	5.75
tower-invert5	9992	1,548.40	**370**	**16.61**	27.01
tower-invert6	$>$ 10,000	–	**1465**	**136.64**	–

Table 4.4: The results on the same Blocks World problems where the goal tasks are specified in the reversed order. A number in boldface indicates the best result (either in CPU time or number of partial plans created) for the problem.

instance, the goals of the tower-invert4 problem used for the above experiments are ordered as n1:(on B C) n2:(on C D) n3:(on D A). The planner using FAF decomposes the goal n1:(on B C) first and and then later decomposes the goal n2(on C D) before decomposing the goal task n3:(on D A).

In order to see if the relative performance of FAF and ExCon changes with the different orderings of the goal tasks, we ran the same test, except that the goals were specified in reverse order. Table 4.4 lists the results. Interestingly, the performance of ExCon was even better for this type of goal specification, while it was far worse for FAF. Notice that the difference between the results of ExCon shown in Tables 4.3 and 4.4 is relatively small; it is never more than double. This is because ExCon chooses tasks based on which task establishes (or threatens) the current applicability condition in the partial plan, and is able to resolve the task interactions efficiently no matter what order the goal tasks happen to be specified.

4.7 Related Work

4.7.1 External conditions in other HTN planners

Nonlin [47], O-Plan2 [10] and SIPE-2 [54] use *condition typing.* In these planners, state constraints are specified with *condition types*, which not only specify the conditions that must be satisfied in the plan, but also specify how the conditions can be used. Using condition typing, the domain writer has more power to control the search for a plan. Some types of conditions in these three planners are defined similarly to the external conditions we use. Since most condition types in Nonlin, O-Plan2, and SIPE-2 are defined and used quite alike, we discuss only the condition types in O-Plan2 in this section. For a summary and comparisons of condition types used in Nonlin, O-Plan2 and SIPE-2, see [48].

In O-Plan2, there are three types of conditions that may satisfy the definition of external conditions. Unsupervised conditions are conditions that are satisfied by other tasks for other goals. Unlike external conditions, however, unsupervised conditions may also specify a condition that can be established by a subtask in the decomposition method. Only_use_if conditions are conditions that are used to filter out inapplicable decomposition methods. If the conditions are for non-static state conditions (i.e. the conditions that may change as results of other actions), then they are considered external conditions by our definition. Only_use_for_query conditions are to bind variables. Similarly to only_use_if conditions, some conditions may be considered external conditions.

Even though these conditions are similar to external conditions, they are used quite differently in these planners, compared with how external conditions are used in our ExCon strategy. For example, if a condition is specified as a unsupervised condition in O-Plan2, the planner assumes that the condition is satisfied by some other task. Thus, satisfying the condition has a low priority during the planning process. On the other hand, satisfying any external condition has a high priority in the ExCon strategy since the planner cannot make such assumptions.

Automatic Extraction

As opposed to condition types explicitly specified by the domain writer, our implementation of ExCon automatically extracts external condition from the domain. Comparing the two approaches, each one has advantages and disadvantages.

The explicit specification of the condition types allows a domain expert more power to control the search as well as defining the application domain. Since a domain expert naturally has the knowledge as to how plans can be constructed efficiently, this approach can make the planning highly efficient. Also, it is currently not possible to extract all the external conditions as mentioned in Section 4.3.1. It is possible that external conditions not extracted may help prune the search furthermore. Meanwhile, it is highly possible that some external conditions as defined in this paper do not help the pruning.

However, modeling and specifying an application domain to work correctly for a planning system requires a lot of effort on the side of a domain expert. Specifying the domain so that the planning would be efficient requires even further efforts and deep understanding of how the planning algorithm works. Our method of preprocessing the knowledge base to extract interesting conditions makes it easier for domain experts to maintain the domain.

4.7.2 High-level effects

Some HTN planners use 'primary effects' of non-primitive tasks (1) to establish conditions of other non-primitive tasks, and (2) to prune partial plans where a condition establishment is threatened by them. Such primary effects are also called 'high-level effects' to distinguish them from the primary effects of action-based operators used in some literatures. For example, the task (Go ?i ?d) - go from ?i to ?d - may have the high-level effect of (*at* ?d). By using high-level effects, an HTN planner can establish many, or most, conditions specified for non-primitive tasks. However, using high-level effects has some drawbacks as well as benefits. In this section, we discuss the use of high-level effects in HTN planning and how our ExCon strategy would perform.

Soundness and completeness

Some HTN planners use high-level effects in a way that makes the planning unsound and/or incomplete.

Using high-level effects can threaten the soundness of HTN planning if one of the following situations occurs:

(a) A high-level effect e associated with a task t_e does not appear in some decompositions of t_e.

(b) A high-level effect e associated with a task t_e is inserted by a task in some decompositions of t_e, but removed by another task.

(c) A high-level effect e associated with a task t_e is inserted by a descendent of t_e but clobbered by an action in the decompositions of other tasks.

Using high-level effects can threaten the completeness of HTN planning if the following situation occurs:

(d) If a high-level effect e associated with a task t_e is used to establish a condition for another task t_c, most planners which use high-level effects usually put an ordering from the end of t_e to the beginning of t_c. However, such an ordering excludes these plans which have t_e after the action that gives e in the decomposition of t_e but before the last action for t_e.

There are two ways which have been suggested on how to incorporate high-level effects into the HTN planning formalism in a way that does not threaten the properties of the planner such as soundness and completeness. One way is to require the domain expert to specify the domain such that for each high-level effect e associated with a task t_e, every decomposition of t_e must contain a subtask with the effect e, which is not clobbered by any other subtask in the same decomposition [55]. A similar approach is used in the DPOCL planner by [57]. Although this approach solves situations (a) and (b) above, it does not solve situation (c) nor (d). Another way is to impose constraints on decompositions such that a planner weeds out the sub-plans that do not give the intended effects. So situations described by (a) or (b) never occur. In most HTN formalisms that use this approach, condition establishments using high-level effects are also protected by similar constraints to avoid situation (c). This approach is used in the UMCP formalism [11] and hybrid planning by [24]. In UMCP, high-level effects cannot be used to fully establish conditions in order to avoid situations described by (d); only a primitive action can establish a condition.

The ExCon strategy and high-level effects

The purpose of using high-level effects is to allow the planner to construct more consistent partial plans at higher levels of abstraction and to reduce the chance of backtracking later because of decisions made for these non-primitive partial plans.

As described above, the UMCP planner does not use high-level effects to fully establish conditions, making backtracking cost bigger than in other planners that use high-level effects to establish conditions. However, the ExCon strategy can compensate for such inefficiency by reducing the backtracking cost incurred by not using high-level effects.

Furthermore, we believe the ExCon strategy could also improve the efficiency of an HTN planner that uses high-level effects to establish conditions. Here are the reasons:

- High-level effects can specify only certain effects of a non-primitive task. Many conditions have to be established by effects not specified as high-level effects.[5] Establishing these conditions may involve interleaving tasks. ExCon can reduce the backtracking costs in such cases.

- Using high-level effects to establish conditions does not mean every condition can be established immediately. The planner still may need to decompose tasks until it finds a task with a high-level effect that can establish the condition. ExCon can identify which task to decompose in such cases.

[5]Some call them the *side effects* of a task.

4.8 Summary

Although FAF helps the planner reduce the size of the search space, it can be outperformed by a strategy that recognizes the opportunities to prune the search. For multi-goal problems, a lot of backtracking cost can be incurred by not recognizing the futility of interleaving tasks in the partial plan early during the planning process. The ExCon strategy can greatly reduce such backtracking costs by detecting and completing task interleaving as soon as possible. The strategy does this by keeping track of external conditions.

In the empirical studies, ExCon outperformed FAF on complex problems, doing increasingly well on problems where the task interactions occurred recursively and where multiple goals were involved. However, ExCon did not outperform FAF on two points:

1. ExCon can be outperformed by FAF for reasons not directly related to task interactions. For example, on 2pack(a) problems in UM Translog, FAF outperformed ExCon because of a big backtracking cost on a variable instantiation. This suggests that we need to investigate a whole refinement strategy instead of focusing on task selection decisions.

2. Since ExCon is designed to do well on problems where interleaving tasks fails, it does not do as well on problems where such failures do not occur. For example, there is little difference between the results of FAF and ExCon on 2-goal 1-predicate problems in the artificial domain. However, we can improve ExCon on some of such problems by utilizing ordering constraints specified for tasks. This method is described in the next chapter.

Since ExCon enables the planner to establish conditions in the plan at less detailed levels, it produces some of the same improvements in planning efficiency that one might try to get using planning constructs such as high-level effects. In addition, it has the following advantages:

- External conditions do not have to be specified explicitly by the user, but instead are found automatically by the planning system when it pre-computes its knowledge based. This will make it much easier for users to maintain the knowledge base.

- ExCon is a task selection strategy, not a search-space pruning heuristic: it simply specifies the order in which a planner will prefer to expand tasks. Thus, it has no effect on the planner's soundness and completeness: a planner that is sound and complete without it will also be sound and complete with it.

Chapter 5

Left-to-Right Method

The ExCon strategy presented in the previous chapter reduces the backtracking costs that are caused by failing to establish applicability conditions by interleaving tasks. Another way to resolve task interactions efficiently is to simply plan the tasks in an order similar to the one used when tasks are executed. By doing so, the planner can easily follow the changes the execution of the tasks make in the state and thus be able to establish many state constraints associated with the tasks that come later. This chapter presents the Left-to-Right heuristic that selects tasks to decompose in a left (the initial state) to right (the final state) manner.

5.1 Forward HTN planning

Some planning applications use forward HTN planning. For example, Smith, et al. [44] cite two such practical planning application domains, computer bridge and microwave module process planning. Although both domains have a lot of step ordering constraints and the planner gets the complete initial state, they have to handle additional requirements: the bridge program has to deal with imperfect information of not knowing what cards each opponent has, and the microwave module process program has to interact with external information sources and do numeric computation. For planning, they use total-order forward HTN planning which is shown to be more suitable than the traditional backward planning approach. Their reason for doing forward planning is that it can reduce the complexity caused by the additional requirements. By doing total-order forward HTN planning, the planner can explicate the current world state a lot more easily since it knows the complete initial state of the world and all the action steps that are going to be in the plan up until the state. If either program used a backward-chaining planning approach, any number of actions could be inserted anywhere in the plan. Thus, it would be extremely difficult to evaluate or validate a partial plan during the planning process. For these reasons, even for domains that do not require numeric computation or do not have to handle imperfect information, it still may be a good idea to do forward planning.

In action-based planning, there are three ways to plan actions: plan forward, plan backward and a combination of both. Constructing plans starting from the initial state gives the planner the advantage of having more information about the world state the planner is dealing with and thus makes it easier to solve interactions between actions. Planning backward from the goals has the advantage of producing lower branching factors because there are usually fewer actions applicable to satisfy a goal than a state. Although both the approaches of forward planning and bi-directional planning have been used successfully by some planners [16, 7], the backward planning approach has been the most popular. For HTN planning, the backward planning approach does not have an obvious advantage since the planner constructs plans by decomposing tasks into subtasks by applying decomposition methods. So planning backward will not reduce the branching factors. On the other hand, the forward planning approach has an advantage similar to action-based planning. First, like action-based planning, the initial state is a complete description of the world state. Since only actions can affect the world state, inserting actions starting from the initial state can provide more state information that is useful in reasoning about later actions. Furthermore, an HTN domain can contain explicit step orderings between subtasks, which make it easier for the planner to select earlier tasks.

5.2 Left-to-Right heuristic

In HTN planning, the Left-to-Right (LtoR) task selection strategy will decompose a non-primitive task only when there are no other non-primitive tasks ordered to come before it. As a tie-breaking rule to handle the case where more than one non-primitive task has only primitive tasks ordered before it, we use the *Fewest Predecessors* (FP) heuristic, which selects non-primitive tasks which have the least number of tasks ordered before them. This heuristic has the advantage that it does not have to check if the preceding tasks are primitive or not because a non-primitive task A has fewer tasks ordered before it than a non-primitive task B if A precedes B. So using the FP heuristic automatically implements LtoR and each computation takes polynomial time with respect to the number of tasks in the partial plan.

As mentioned in the previous chapter, one class of problems where ExCon does not perform well is the one where the no tasks can be interleaved. The planner cannot interleave non-primitive tasks in a problem if (a) the tasks in the problems are independent of each other, and/or (b) the tasks in the problem are totally ordered. For the problems of type (a), decisions made for one task do not affect decisions that must be made for another task. Thus, we conjuncture that the choice of task selection method will have little effect on the planning efficiency. For the problems of type (b), LtoR should do well for the reasons stated above. Therefore, if we use LtoR as a tie-breaker in ExCon, the resulting ExCon strategy should do well both on problems where the tasks can be interleaved and on problems where the tasks are totally ordered.

While LtoR has an advantage similar to forward planning in action-based planning, the LtoR strategy we present does not necessarily "plan forward" unless the goal tasks and their subtasks are totally ordered. For example, LtoR may select to decompose a non-primitive task A before it selects a non-primitive task B, yet the subtasks of A may be ordered after the subtasks of B as a result of satisfying some state constraints. An alternative way is to use the algorithm Nau, et al. [34] suggest, which commits to a certain ordering between non-primitive tasks when decomposing tasks.

5.2.1 Implementation

We implemented LtoR[1] and another version of ExCon with LtoR as a tie-breaker. In order to distinguish between the two versions of ExCon with different tie-breakers, we call them ExCon-FAF and ExCon-LtoR for ExCon with FAF and ExCon with LtoR, respectively.

[1]We used FAF as a tie-breaker for LtoR.

5.3 Experiments

In order to see how the task selection methods affect the planning efficiency on problems with different amounts of ordering constraints and different amounts of tasks that can be interleaved, we ran FAF, LtoR, ExCon-FAF and ExCon-LtoR on a small artificial domain and UM Translog. We hypothesize the following:

- Similarly to ExCon-FAF and FAF, we hypothesize that the ExCon-LtoR strategy should outperform the LtoR strategy especially on problems where the goal tasks are highly interleaved.
- In planning domains in which there are many constraints on the ordering of the subtasks, the LtoR heuristic should be able to outperform the FAF heuristic by expanding the tasks in an order that facilitates the pruning of infeasible plans. Similarly, the ExCon-LtoR strategy should be able to outperform the ExCon-FAF strategy in such problem domains.

5.3.1 Random Travel Planning domain

In our description of the LtoR strategy earlier, we pointed out that the explicit step orderings given in the task descriptions make it easier for the planner to select the tasks in a left-to-right manner. To see how much this can help the performance of the planner, we created a small domain called Random Travel Planning where there is only one level of hierarchy (i.e. no non-primitive task is decomposed into another non-primitive task). In this domain, there are three types of goal tasks, Sightsee, Travel and Eat. Their decomposition methods are shown in Figure 5.1.[2]

[2]The domain is slightly different from the one we used in [49] since the change from UMCP1.0 to UMCP1.2 made the problems on Random Travel Planning domain easier to solve for most of

The task Sightsee is to go sightseeing ourselves (Method 1 in the figure) or to join a tour bus (Method 2), depending on if we are tired or not. Going sightseeing makes us tired. Taking a flight, or eating a food makes us recover from tiredness. In other words, the primitive task (go-sightseeing ?city) has an effect (*tired* ?city) and the primitive tasks (fly ?city0 ?city1) and (have ?food ?city0) have an effect ($\sim$*tired* ?city0). The task (fly ?city0 ?city1) also has effects ($\sim$*in* ?city0) and (*in* ?city1). Other primitive tasks have no effects. The task Travel is to move to another city, if it's different from the current location. The task Eat is to go to a restaurant for a type of food we want and eat there. If the type of food we want is local to the location, such as Italian food if the current location is Rome, then going to a local restaurant suffices (Method 1). If not, we go to a good restaurant if we are not tired (Method 2), or we go to a closer restaurant if we are (Method 3). In these methods, every 'before' and 'between' state constraint is an external condition.

A problem in this domain consists of 10 goal tasks, randomly generated. A goal task is either (Sightsee), (Travel ?city), or (Eat ?food). where ?city is a value randomly chosen from {LosAngeles, NewYork, London, Paris, Rome}, and ?food is a value randomly chosen from {American-food, English-food, French-food, Italian-food}. Since the subtasks in each method are totally ordered, how the problems are ordered depends on the step orderings between the goal tasks. If the goal tasks are totally ordered, then every partial plan generated from the goal also has the tasks totally ordered. The step orderings between the goal tasks are randomly generated based on the parameter ω. ω defines the maximum number of pairs of goal tasks that can be left unordered. Lower values of ω indicate that there are more ordering constraints among the goal tasks. The initial state consists of

the problems.

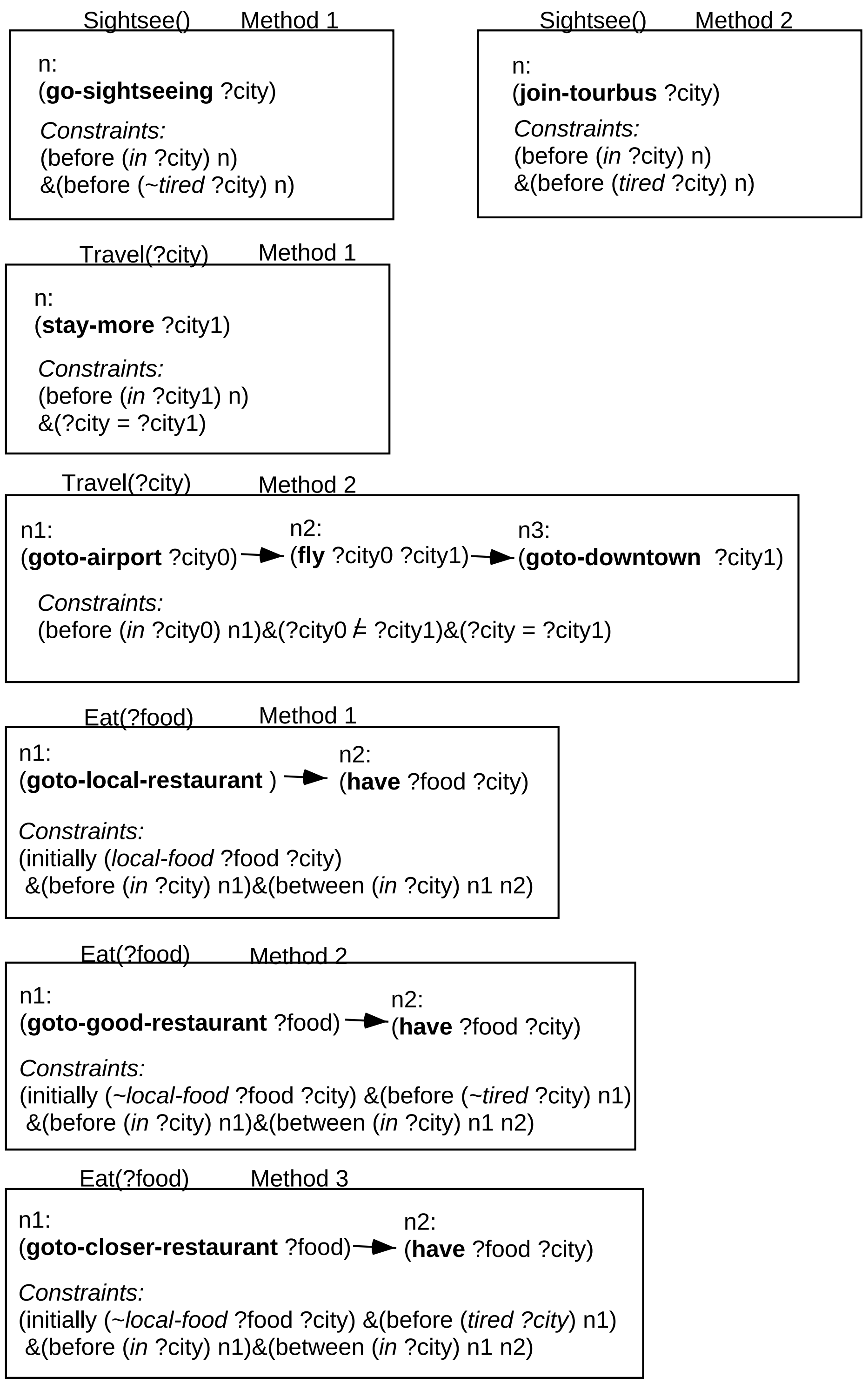


Figure 5.1: The decomposition methods for the Random Travel Planning domain. The tasks shown in boldface are primitive tasks.

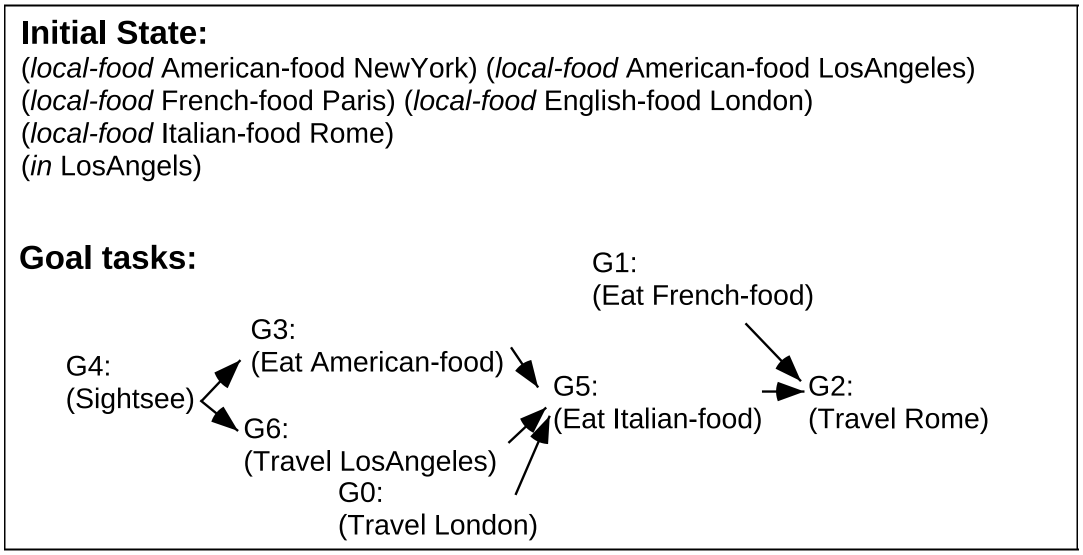
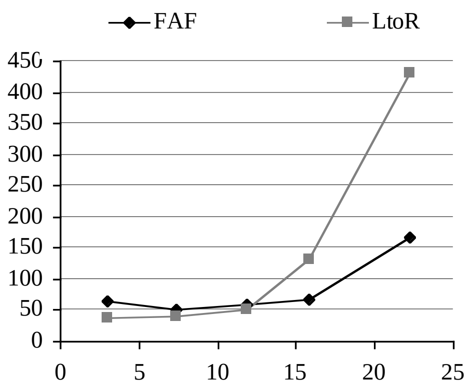


Figure 5.2: A problem of 7-goals, $\omega = 10$ (the actual number of unordered pairs of task is 9) in the Random Travel Planning domain.

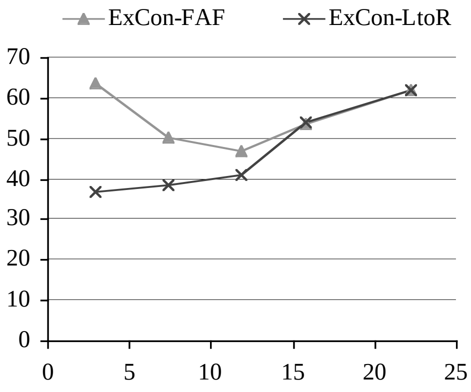
the food-city pairs for each city such as (*local-food* Italian-food Rome) and the current location, i.e. (*in* ?city), which is randomly assigned. A sample 7-goal problem of $\omega = 10$ is shown in Figure 5.2

We created 20 10-goal problems each for ω = 5, 10, 15, 20 or 25 and solved them using FAF, LtoR, ExCon-FAF and ExCon-LtoR strategies. The results are shown in the Table 5.1 and Figure 5.3. For low ω values, LtoR does better than FAF because LtoR can use the step orderings to correctly choose the earliest tasks. Many applicability conditions considered by ExCon-LtoR can be easily established at the time the conditions are inserted into the plan by using LtoR selection. Also, there are fewer non-primitive tasks that may affect the establishment of the current applicability condition, so the performance of ExCon-LtoR is similar to that of LtoR.

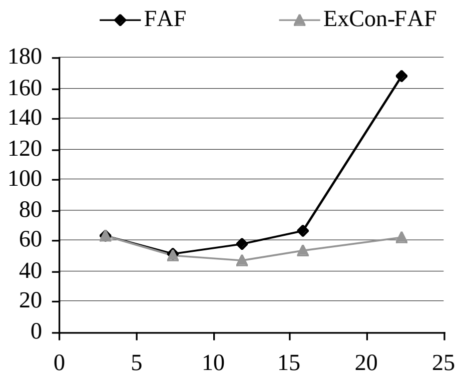
The performances of FAF and ExCon-FAF are also similar for the low ω value problems. Since FAF uses the LtoR heuristic for tie-breaking, FAF picks up the



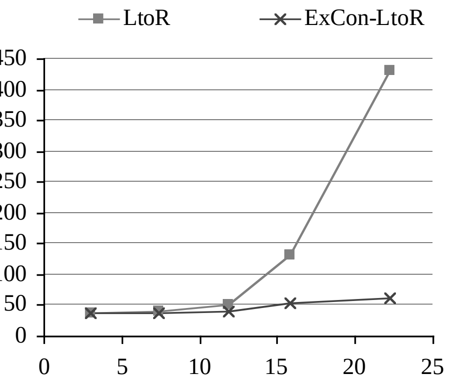
(a) FAF and LtoR



(b) ExCon-FAF and ExCon-LtoR



(c) FAF and ExCon-FAF



(d) LtoR and ExCon-LtoR

Figure 5.3: The results of the Random Travel Planning problems. The x-axis shows the average number of pairs of unordered goal tasks and the y-axis shows the average number of partial plans created.

ω	Actual	FAF		LtoR		ExCon-FAF		ExCon-LtoR	
		Plans	Time	Plans	Time	Plans	Time	Plans	Time
5	2.95	63.55	0.33	**36.95**	**0.16**	63.60	0.36	**36.90**	0.18
10	7.35	51.25	0.28	41.00	0.21	50.45	0.30	**38.40**	**0.20**
15	11.85	58.55	0.34	50.80	0.27	46.90	0.28	**41.05**	**0.22**
20	15.8	66.65	0.49	131.85	1.23	**53.75**	**0.35**	54.15	**0.35**
25	22.2	168.10	2.84	431.55	6.20	62.15	0.45	**62.00**	**0.42**

Table 5.1: The results of the Random Travel Planning problems. The actual average number of unordered pairs is shown next to ω values. Plans is the average number of partial plans created. Time is average non-garbage collection CPU time in seconds.

tasks relatively from left-to-right, although it skips Eat in preference to Sightsee or Travel. So, similarly to ExCon-LtoR, only a few possible non-primitive tasks exist that may affect the establishment of the current applicability condition. Since the performance of the ExCon strategy greatly depends on its tie-breaking strategy for low ω value problems (i.e. problems where goal tasks are not interleaved much), ExCon-LtoR does better than ExCon-FAF.

For the high ω value problems, there are not very many ordering constraints among the goal tasks, so there can be many more interactions among goal tasks. In these problems, the performance of LtoR is worse than any other strategy because LtoR does not have enough step ordering information to correctly work in a left-to-right manner and thus cannot identify constraints that can never be established early. FAF performs better than LtoR, but not as well as ExCon-FAF or ExCon-LtoR. The performances of ExCon-LtoR and ExCon-FAF are similar because for

ω	5	10	15	20	25
ExCon-FAF/ExCon-LtoR	1.72	1.31	1.14	0.99	1.00
p	96.3%	81.6%	94.7%	40.4%	51.2%

Table 5.2: Comparison of ExCon-FAF and ExCon-LtoR on the Random Travel Planning problems.

problems with highly interleaved goal tasks, ExCon selects tasks and does not have to use its tie-breaking strategy (i.e. LtoR or FAF).

We also performed a one-tailed paired t-test on the Random Travel Planning results to compare the performance of ExCon-FAF and ExCon-LtoR. Similarly to the test in the previous chapter, the null hypothesis is $H_0 : \mu_{ExCon-FAF} = \mu_{ExCon-LtoR}$ and the counter hypothesis is $H_1 : \mu_{ExCon-FAF} > \mu_{ExCon-LtoR}$, where $\mu_{ExCon-FAF}$ and $\mu_{ExCon-LtoR}$ are the mean numbers of partial plans created by using ExCon-FAF and ExCon-LtoR, respectively. The resulting p values are shown in Table 5.2.

For $\omega = 5$, we can reject the null hypothesis with a confidence level 96.5%. For $\omega = 10$ and 15, the confidence level is lower. And for $\omega \geq$ 20, the p-value shows that neither of the two strategies is significantly doing better than the other. Thus, ExCon-LtoR outperforms ExCon-FAF on problems with many ordering constraints, while their performances are similar on problems where more tasks can be interleaved.

5.3.2 UM Translog

Next, we looked at the performance of the task selection methods on problems in the UM Translog domain. For one-package problems in the UM Translog domain,

Problem	LtoR		ExCon-FAF		ExCon-LtoR		E-F/E-L	p
	Plans	Time	Plans	Time	Plans	Time	(Plans)	
1pack	**73.40**	**0.30**	76.00	0.35	**73.40**	0.37	1.04	>99.9%

Table 5.3: The results for LtoR, ExCon-FAF and ExCon-LtoR for one-package UM Translog problems. Plans is the average number of partial plans created. Time is non-garbage collection CPU time in seconds. p represents the confidence levels of paired t-test for ExCon-FAF and ExCon-LtoR.

almost all the subtasks of a goal task are ordered with each other. Hence, we hypothesized LtoR should do well for these problems. For multiple-package problems, there are no ordering constraints. So, LtoR should do worse for these problems. However, ExCon-LtoR should do well for the same reason that ExCon-FAF did well on these problems. We ran the same UM Translog problems we used in the previous chapter. For one-package problems, LtoR, ExCon-FAF, and ExCon-LtoR were tested. For multiple-package problems, only ExCon-FAF and ExCon-LtoR were tested because LtoR performed far worse than any other strategies.

One-package problems

Table 5.3 shows small differences between the performances of three strategies on one-package problems in Translog. In terms of the number of partial plans created, LtoR and ExCon-LtoR constantly outperformed ExCon-FAF. However, in terms of CPU time, LtoR performed best although FAF sometimes outperformed it. This is probably because FAF needs less computation overhead to compute which task has the fewest decomposition methods. On the other hand, the ExCon strategy needs more overhead to keep track of external conditions and figure out which task can establish which constraint. Thus, even though ExCon-LtoR did well in terms of the number of partial plans, it did not do so in terms of CPU time.

The table also shows the results of the t-test used to compare the result of ExCon-FAF and ExCon-FAF using the number of partial plans. Since ExCon-LtoR constantly outperformed ExCon-FAF in terms of the number of partial plans, we can say that ExCon-LtoR creates less partial plans than ExCon-FAF with a confidence level of 99.9%.

Multiple-package problems

Table 5.4 shows the results of two- and three-package problems for ExCon-FAF and ExCon-LtoR. We did not show LtoR for these problems because it performed significantly worse than other strategies. For example, LtoR created 2129.6 partial plans on average for the 2pack(a) problems.

The table shows the mixed results: in terms of the number of partial plans, ExCon-FAF outperformed ExCon-LtoR on 4 problem types (2pack(a),(c)-(d) and 3pack(a)) and ExCon-LtoR outperformed ExCon-FAF on the remaining 4 problem types (2pack(b), 3pack(b)-(d)). For problems with the same itinerary (i.e. 2pack(a) and 3pack(a)), ExCon-LtoR is outperformed by both FAF (shown in Table 4.2) and ExCon-FAF, although ExCon-LtoR outperformed FAF for all the other problem types. Since the planner can successfully interleave goal tasks for these problems, ExCon does not have the advantage. Thus, the performance of ExCon depends on its tie-breaking strategy. Since there are no ordering constraints between goal tasks in multiple package problems, the LtoR heuristic does not have an advantage. Thus, ExCon-FAF outperforms ExCon-LtoR.

For 2pack(b)-(d), the difference between the ExCon-LtoR and ExCon-FAF is small, but the t-test shows it is significant. Which strategy is better varies

Problem	Types	ExCon-FAF		ExCon-LtoR		E-F/E-L	p
		Plans	Time	Plans	Time	(Plans)	
2pack(a)	same	**178.20**	**1.98**	179.90	**1.98**	0.99	2.5%
2pack(b)	same	229.65	3.04	**227.55**	**2.82**	1.01	99.8%
2pack(c)	same	**536.25**	10.37	551.10	**10.12**	0.97	<0.01%
2pack(d)	different	**633.40**	**9.90**	672.35	10.11	0.99	<0.01%
3pack(a)	same	**238.15**	**6.28**	814.90	26.77	0.29	<0.01%
3pack(b)	same	33639.15	1,075	**18657.70**	**687**	1.80	>99.9%
3pack(c)	same	57692.10	1,937	**33053.50**	**1,428**	1.75	>99.9%
3pack(d)	different	20733.55	556	**13087.75**	**419**	1.58	>99.9%

Table 5.4: The results of ExCon-FAF and ExCon-LtoR for multiple-package problems in UM Translog. Plans is the average number of partial plans created. Time is non-garbage collection CPU time in seconds. "Types" shows whether the packages have the same type and so can be carried by the same delivery truck. p represents the confidence levels of paired t-test.

between problem sets. For 3pack(b)-(d), the difference is bigger, and ExCon-LtoR outperformed ExCon-FAF. These results show that the tie-breaking strategy plays a big role in ExCon. Since the tie-breaker determines which task to decompose at the start of the planning process, it also determines which applicability conditions ExCon will first look at. We need to further investigate which tie-breaking strategy should be used for highly interleaved problems.

5.4 Summary

This chapter investigated LtoR, a task selection heuristic that takes advantage of ordering constraints in the domain. If the planner knows the order in which tasks are later executed, it can plan efficiently by planning in the same order because it can establish most state constraints associated with tasks immediately by examining the changes made in the world state by the tasks that come before them. We implemented LtoR, which selects tasks to decompose if there are no non-primitive tasks ordered before them.

We have compared LtoR, FAF, ExCon-FAF and ExCon-LtoR on two test domains. The primary results are as follows:

- LtoR and ExCon-LtoR perform better than FAF and ExCon-FAF on problems where there are many ordering constraints.
- LtoR performs far worse than any other strategies on problems where there are less ordering constrains.
- ExCon-FAF and ExCon-LtoR perform better than FAF and LtoR on problems where there are less ordering constraints and tasks can be interleaved.

The above results seem to show that ExCon-LtoR is preferable to ExCon-FAF because it can perform well on problems where there are a lot of ordering constraints (i.e. LtoR performs well) as well as on problems where many tasks can be interleaved (i.e. ExCon performs well). However, the results also show that the range of problems that LtoR can perform well on is quite limited. LtoR and ExCon-LtoR performed better than FAF and ExCon-FAF for problems of $\omega = 5$ and 10 in the Random Travel Planning domain. Still, the t-test applied to the results of ExCon-FAF and ExCon-LtoR can only reject the null hypothesis at a confidence level of 96.5% for $\omega = 5$, a rather low level. This implies that the difference between ExCon-FAF and ExCon-LtoR even on problems with $\omega = 5$ is not that significant. Considering that the actual average ω value is 2.95, the goal tasks in those problems are almost linearly ordered. Thus, the advantage presented by using the LtoR heuristic applies only to a small range of problems. Even though, we believe the LtoR heuristic would do well on many real-world problems because there are many domains that have only linearly ordered tasks, as Smith, et al. [44] suggest.

Although the experiments on LtoR and ExCon-LtoR showed that LtoR performs well on problems where tasks are almost linearly ordered, there remain questions regarding the four strategies tested:

- ExCon-FAF and ExCon-LtoR perform better than FAF and LtoR on problems where many tasks can be interleaved. However, the tie-breaking strategy seems to affect the performance of ExCon. We need to study further what tie-breaking strategy is best for ExCon for what types of problems.

- Both of the results of experiments in the Translog domain and the Random Travel Planning domain seem to indicate that LtoR does perform well on problems where subtasks have many ordering constraints between them. However, the savings from using LtoR are not large compared with FAF or ExCon-FAF in either of the two domains. This may be due to the fact that FAF uses the LtoR heuristic to break ties. Further investigation is necessary to fully evaluate how well LtoR does compared with FAF or ExCon-FAF.

Chapter 6

Conclusion

6.1 Research Contributions

The goal of this dissertation was to analyze refinement strategies for HTN planning and address issues on the efficiency of HTN planning. In each analysis, strategies were evaluated systematically: they were tested both on artificial domains as well as on more realistic domains, and statistical tests were performed on the results. The primary research contributions of this work are summarized below.

6.1.1 Problems with Least Commitments

Most refinement planning systems use some type of least commitment strategy, which tries to delay commitments to certain elements of a plan in order to avoid making premature commitments. Many studies on least commitments show that the least commitment strategy is better than other previous strategies. A problem exists, however, in choosing which commitments should be delayed. Since a planner has to commit to something in order to plan, it cannot delay committing to every element of a plan. Thus, a least commitment strategy may make premature decisions by delaying some other decisions.

I compared three strategies, RVBS (least commitment to variable instantiation), EVIS (least commitment to task instantiation), and DVCS (a dynamic strategy which chooses between RVBS and EVIS using the FAF heuristic). The results were the following:

- A least commitment strategy which consistently delays commitments to certain elements of a plan throughout the planning process can make premature commitments to other elements of a plan.
- The decision of which elements of a plan to delay committing to should be altered during the planning process, depending on the current partial plan.
- The FAF heuristic can be used to make such decisions. In the experiments of the two least commitment strategies, RVBS and EVIS, and of DVCS, I have shown that neither RVBS nor EVIS can perform best in all types of problem domains. On the other hand, DVCS performed better than or as well as RVBS and EVIS in all of our test domains by choosing between the two strategies using the FAF heuristic.

6.1.2 AND/OR Serialization

During refinement search, a refinement strategy needs to decide which refinement to do among many refinements that are applicable to the current search node. This choice process that a refinement strategy makes can be viewed as the process of serializing an AND/OR graph into an OR-tree. Thus, the performance of a refinement strategy can be evaluated by the size of the serialized tree it makes. While this evaluation ignores the effects of pruning that a refinement strategy can make during the search process, it provides a good estimate of the size of search space in the worst case.

The difference between the best possible serialization and worst possible serialization can be as big as the factor of 2^k where $2k$ is the height of the AND/OR tree. Also, in this model, the best serialization can be obtained by using the FAF heuristic. In general, the FAF heuristic does not always generate the best serialization; in fact, it can generate the worst serialization if given the right AND/OR tree. However, the experiments using randomly generated AND/OR trees show FAF generated the best or near-best serialization in most cases.

6.1.3 Task Interactions in HTN Planning

The various analyses of FAF indicate that it works well on many problems. However, since FAF does not prune the search space, a refinement strategy that effectively prunes the search space can outperform FAF.

In HTN planning, the planner interleaves non-primitive tasks in order to generate a plan that contains less redundant actions. In order to do so, the planner typically tries to interleave tasks as much as possible. However, in many cases, the planner cannot interleave tasks for reasons such as that the task does not produce the necessary effect or that some other task interferes with the necessary effect. The ExCon strategy tries to detect and deal with task interleaving as early as possible, in order to prune the search space when it finds the task interleaving futile. The empirical results on ExCon and FAF show that ExCon increasingly outperformed FAF on problems where the planner fails to interleave tasks.

6.1.4 Ordering Constraints in HTN Problems

Unlike action-based planning, the user can provide ordering constraints between tasks in an HTN planning domain. An HTN planner can prune the search space by using such ordering constraints to limit orderings between tasks. The LtoR strategy expands the tasks in an order similar to the order that the tasks are later executed. If the tasks for the given problem have many ordering constraints, LtoR can establish most state constraints immediately after they are asserted into the plan.

In a comparison of LtoR, FAF, ExCon-FAF (ExCon with FAF as a tie-breaker) and ExCon-LtoR (ExCon with LtoR as a tie-breaker), the results show the following:

- Both LtoR and ExCon-LtoR outperformed FAF and ExCon-FAF on problems where there are ordering constraints between almost all the tasks. However, the range of problems where LtoR can perform well is quite limited.
- ExCon-LtoR and ExCon-FAF outperformed FAF and LtoR on problems where there are less ordering constraints and tasks can be interleaved.

6.2 Future Research Directions

The work described in this dissertation suggests several topics for future work:

- **Improving ExCon** The study of the two versions of ExCon (ExCon-FAF and ExCon-LtoR) shows that the tie-breaker can make a big difference on the performance of ExCon. Thus, an extensive study of the tie-breaking strategy for ExCon is needed to improve ExCon. Also, it is not clear in what order external conditions should be looked at in ExCon to improve the efficiency. For example, if a decomposition method has more than one external condition, which external condition should be considered first by ExCon? Currently, external conditions are considered in the order that the planner extracts them. It is necessary to examine how much the order in which external conditions are considered during the planning process affects the performance.

- **Combining refinement heuristics** On some problems, ExCon's performance is worse than FAF. This is due to wrong decisions for variable bindings rather than the decision of task selection. However, this suggests an analysis of the whole refinement strategy is needed instead of focusing on task selection as was done for the studies of ExCon and LtoR. In order to do so, combinations of heuristics for the different refinement planning elements (tasks selection, variable constraints, step orderings) should be evaluated to see if there are any combined effects on the efficiency of planning.

- **Choosing an appropriate refinement strategy for a given problem.** One refinement strategy cannot perform well on every kind of problem. Even when strategy A is shown to do better than strategy B, a minor change to A may make multiple versions of strategy A that may perform differently on different problems as shown by the study of ExCon-FAF and ExCon-LtoR. Thus, each strategy should be evaluated in such a way that shows the conditions of the planning problems under which each strategy performs well. Using such information, the user or the planner can make decisions to choose appropriate refinement strategies based on problem features. This requires a systematic way to categorize problems based on their characteristics.

BIBLIOGRAPHY

[1] M. Aarup, M. M. Arentoft, Y. Parrod, J. Stader, and I. Stokes. OPTIMUM-AIV: A knowledge-based planning and scheduling system for spacecraft AIV. *Intelligent Scheduling*, pages 451–469, 1994.

[2] J. M. Agosta. Formulation and implementation of an equipment configuration problem with the SIPE-2 generative planner. In *Proceedings of AAAI-95 Spring Symposium on Integrated Planning Applications*, pages 1–10, 1995.

[3] Scot Andrews, Brian Kettler, Kutluhan Erol, and James Hendler. UM translog: A planning domain for the development and benchmarking of planning systems. Technical report, Dept. of Computer Science, University of Maryland, College Park, MD, 1995.

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

[5] Anthony Barrett and Daniel S. Weld. Task-decomposition via plan parsing. In *Proceedings of AAAI-94*, pages 1117–1122, 1994.

[6] James Bitner and Edward Reingold. Backtrack programming techniques. *Communications of the ACM 18(11)*, pages 651–656, 1975.

[7] A. L. Blum and M. L. Furst. Fast planning though planning graph analysis. *Artificial Intelligence 90(1-2)*, pages 281–300, 1997.

[8] D. Chapman. Planning for Conjunctive Goals. *Artificial Intelligence 32*, pages 333–377, 1987.

[9] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer, New York, 1981.

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

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

[12] Kutluhan Erol, James Hendler, and Dana Nau. HTN planning: Complexity and expressivity. In *Proceedings of AAAI-94*, pages 1123–1128, 1994.

[13] Kutluhan Erol, James Hendler, and Dana Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of AIPS-94*, pages 249–254, 1994.

[14] Kutluhan Erol, James Hendler, Dana Nau, and Reiko Tsuneto. A critical look at critics in HTN planning. In *14th International Joint Conference of Artificial Intelligence*, pages 1592–1598, 1995.

[15] Richard Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Readings in Planning*, pages 88–97, 1971.

[16] Eugene Fink and Manuela Veloso. Prodigy planning algorithm. Technical report cmu-cs-94-123, Carnegie Mellon University, Pittsburgh, PA, 1994.

[17] Marc Friedman and Daniel S. Weld. Least-commitment action selection. In *AIPS-96*, 1996.

[18] Alfonso Gerevini and Lenhart Schubert. Accelerating partial-order planners: Some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research 5*, pages 95–137, 1996.

[19] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence 56(2-3)*, pages 223–254, 1992.

[20] David Joslin and Martha E. Pollack. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *The Proceedings of 12th National Conference on Artificial Intelligence*, pages 1004–1009, 1994.

[21] David Joslin and Martha E. Pollack. Is "Early Commitment" in plan generation ever a good idea? *Proceedings of AAAI-96*, pages 1188–1193, 1996.

[22] Subbarao Kambhampati. Can we bridge refinement-based and SAT-based planning techniques? In M. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 44–49. Morgan Kaufmann, 1997.

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

[24] Subbarao Kambhampati, Amol Mali, and Biplav Srivastava. Hybrid planning for partially hierarchical domains. In *The Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 882–888, 1998.

[25] Henry Kautz and Bart Selman. Planning as satisfiability. In *The Proceedings of Tenth European Conference on Artificial Intelligence*, pages 359–363, 1992.

[26] Craig A. Knoblock. *Generating Abstraction Hierarchies.* Kluwer Academic Publishers, 1993.

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

[28] Amnon Lotem, Dana Nau, and James Hendler. Applying the graphplan approach to htn planning. In *AIPS-98 Workshop on Planning as Combinatorial Search*, 1998.

[29] Sean Luke. A fast probabilistic tree generation algorithm. Unpublished manuscript, 1997.

[30] Amol Mali and Subbarao Kambhampati. Encoding htn planning in propositional logic. In *The Proceedings of AIPS-98*, page ??, 1998.

[31] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *AAAI-91*, 1991.

[32] D. V. McDermott. Flexibility and efficiency in a computer program for designing circuits. Technical Report AI-TR-402, AI Lab, MIT, 1977.

[33] Steven Minton, John Bresina, and Mark Drummond. Total-order and partial-order planning: A comparative analysis. *JAIR 2*, pages 227–262, 1994.

[34] Dana Nau, Stephen J. J. Smith, and Kutluhan Erol. Control strategies in HTN planning: Theory versus practice. In *The Proceedings of the Tenth Conference on Innovative Applications of Artificial Intelligence*, pages 1127–1133, 1998.

[35] N. Onder and M. E. Pollack. Contingency selection in plan generation. In Sam Steel and Rachid Alami, editors, *Proceedings of the 4th European Conference on Planning (ECP-97): Recent Advances in AI Planning*, volume 1348 of *LNAI*, pages 364–376, Berlin, September24 –26 1997. Springer.

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

[37] Mark A. Peot and David E. Smith. Threat-removal strategies for partial-order planning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 492–499, Washington, D.C., USA, August 1993. AAAI Press/MIT Press.

[38] Martha E. Pollack, David Joslin, and Massimo Paolucci. Flaw selection strategies for partial-order planning. *Journal of Artificial Intelligence Research 6*, pages 223–262, 1997.

[39] Louise Pryor and Gregg Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.

[40] Paul W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence 21*, pages 117–133, 1983.

[41] Earl Sacerdoti. The nonlinear nature of plans. In *The Proceedings of Fourth International Joint Conference on Artificial Intelligence*, pages 206–214, 1975.

[42] David E. Smith and Mark A. Peot. A critical look at knoblock's hierarchy mechanism. In James Hendler, editor, *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS 92)*, pages 307–308, College Park, Maryland, USA, June 1992. Morgan Kaufmann.

[43] Stephen J. Smith, Dana Nau, and Thomas Throop. Total-order multi-agent task-network planning for contract bridge. In *AAAI-96*, pages 108–113, 1996.

[44] Stephen J. Smith, Dana Nau, and Thomas Throop. Success in spades: Using AI planning techniques to win the world championship of computer bridge. In *The Proceedings of the Tenth Conference on Innovative Applications of Artificial Intelligence*, pages 1079–1086, 1998.

[45] Mark Stefik. Planning with constraints. *Artificial Intelligence 16*, pages 111–140, 1981.

[46] George C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12:179–196, 1979.

[47] Austin Tate. Generating project networks. In *The Proceedings of Fifth International Joint Conference on Artificial Intelligence*, pages 888–893, 1977.

[48] Austin Tate, Brian Drabble, and Jeff Dalton. The use of condition types to restrict search in an AI planner. In *The Proceedings of Twelfth National Conference on Artificial Intelligence*, pages 1129–1134, 1994.

[49] Reiko Tsuneto, James Hendler, Dana Nau, and Leliane Nunes de Barros. Matching problem features with task selection for better performance in HTN planning. In *AIPS Workshop on Knowledge Engineering and Acquisition for Planning*, pages 85–93, 1998.

[50] Manuela Veloso and Peter Stone. FLECS: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research 3*, pages 25–52, 1995.

[51] Steven A. Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and machine Intelligence Vol. 5*, pages 246–267, 1983.

[52] Robert Wilensky. A model for planning in complex situations. *Cognition and Brain Theory*, IV(4), 1981.

[53] D. E. Wilkins and R. V. Desimone. Applying an AI planner to military operations planning. *Intelligent Scheduling*, pages 685–709, 1994.

[54] David Wilkins. *Practical Planning.* Morgan Kauffman, 1988.

[55] Qiang Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence 6*, pages 12–24, 1990.

[56] Qiang Yang and Alex Y. M. Chan. Delaying variable binding commitments in planning. In *AIPS-94*, pages 182–187, 1994.

[57] R. Michael Young, Martha E. Pollack, and Johanna D. Moore. Decomposition and causality in partial order planning. In *AIPS-94*, pages 182–187, 1994.