Improving the Efficiency of Planning

by

Qiang Yang

Research Report CS-89-34
August, 1989

Improving the Efficiency of Planning

by
Qiang Yang

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy
1989

Abstract

Title of Dissertation: **Improving the Efficiency of Planning**

Qiang Yang, Doctor of Philosophy, 1989

Dissertation directed by: Dana S. Nau, Associate Professor,
Department of Computer Science

Symbolic planning involves extensive computation for detecting and handling goal interactions. Past planning systems can be classified as either domain-dependent or domain-independent, depending on how they deal with the goal interactions. Domain-dependent systems rely on heuristics that are specific to their particular application domains, and are of limited generality. Domain-independent systems make use of general planning knowledge independent of any particular domains, but have been mainly inefficient.

This thesis presents two approaches to improving the efficiency and broadening the applicability of planning systems. First, a set of conditions are set up upon the representation of planning knowledge. Whenever these conditions are satisfied, efficiency can be provably improved. In addition, these conditions enable one to preprocess the planning knowledge of a system before problem solving starts, and offer guidelines for the design of knowledge representation. Second, a set of restrictions are imposed on the goal interactions so that, whenever they are satisfied, efficient planning methods can be developed. The restrictions does not depend on any specific domain knowledge. Therefore the approach is more general than domain-dependent planning. In addition, they offer more knowledge about goal interactions and thus enable more efficient methods to be developed than purely domain-independent approaches.

Acknowledgements

I wish to express my gratitude to my advisor, professor Dana S. Nau, for his guidance, encouragement, and support during my graduate studies. He introduced me to the field of automated manufacturing, and encouraged me to expand my ideas in that area. I would also like to thank professor James A. Hendler, with whom I had many enlightening discussions which eventually led to some of the major parts in this thesis.

I am indebted to many of my friends and colleagues in the department. Paul Chi not only aided me in my transfer from astronomy to computer science, but also introduced me into Dana's research group. George Vanecek, Raghu Karinthi, Subbarao Kambhampati and Scott Thompson gave me enormous encouragement and help during the writing of my thesis. I would also like to thank Josh Tenenberg in U. of Rochester, whose comments on my earlier writings helped me greatly.

I would like to thank my parents and my sister for their continuous support and inspiration throughout my studies.

My greatest "thank you" I reserve for my wife, Jill Yang, whose love, support and desire to escape Washington,DC spurred me on to the completion of this work.

This work was supported in part by NSF Presidential Young Investigator award DCR-83-51463, with matching funds provided by Texas Instruments and General Motors Research Laboratories, and by NSF Engineering Research Centers Program Grant NSFD CDR-88003012 to the University of Maryland Systems Research Center.

To my immediate family:

my parents, Xiu-ying Li and Hai-shou Yang,

my sister Yuan Yang,

and my wife Jill Yang.

Contents

1 **Introduction** **1**

1.1 Overview 2

2 **Background** **5**

2.1 Goal Interactions in Planning 5

2.2 Domain-Independent Planning 8

2.3 Domain-Dependent Planning 10

2.4 Motivation 11

I **Improving the Efficiency of Hierarchical Planning** **13**

3 **Hierarchical Planning—Definitions** **14**

3.1 Representation 15

3.2 Planning with Action Reductions 19

3.3 Overview 22

4 **The Treatment of Unresolvable Conflicts** **29**

4.1 Motivation 29

4.2 Plan Unlinearizability 32

4.2.1 Definitions 32

4.2.2 Imposing Syntactic Restrictions 33

4.3 Downward-Unlinearizability 39

4.3.1 The "Unique Main Subaction" Restriction 39

4.3.2 Checking for Downward-Unlinearizability 44

4.3.3 Modifying Action Reduction Schemata . . . 45
4.4 Preprocessing Individual Action Template . . . 48
4.5 Augmented Reduction of Protection Intervals . . . 49
4.6 Point-Protected Conditions . . . 54
4.6.1 Definitions . . . 54
4.6.2 Strong Unresolvability . . . 55
4.7 Other Kinds of Goal Interactions . . . 58

5 Ordering-Induced Independence **60**
5.1 Motivation . . . 60
5.2 Imposing Syntactic Restrictions . . . 62
5.3 Checking for "Ordering Induced Independence" Property . . . 64
5.3.1 A Special Case Of Schemata Definition . . . 66

II Multiple Goal Plan Optimization with Limited Interactions **68**

6 Planning For Multiple Goals—Definitions **69**
6.1 Problem Statement . . . 69
6.2 Types of Inter-Goal Interactions . . . 71
6.3 The Detection of Goal Interactions . . . 76

7 One Plan For Each Goal **79**
7.1 Complexity . . . 79
7.2 Plan Existence . . . 80
7.3 Plan Optimality with Restrictions . . . 82
7.3.1 A Branch-and-Bound Algorithm . . . 83
7.3.2 Empirical Results . . . 87
7.4 Partial Ordering Restriction . . . 88

8 More Than One Plan For Each Goal **92**
8.1 Complexity . . . 93
8.2 A Heuristic Algorithm . . . 94
8.3 Experimental Results . . . 102

9 Preprocessing Search Spaces for Single Goal Planning **106**

9.1 Introduction 106

9.2 Problem Characteristics 107

9.3 Threaded Decision Graphs 108

9.4 Search with the Threaded Decision Graphs 111

9.5 The Construction of Threaded Decision Graphs 112

9.6 Very Large or Infinite Search Spaces 113

9.7 Discussion 113

10 Conclusion **116**

10.1 Summary 116

10.2 Limitations And Future Work 118

10.2.1 More Expressive Action Representations 118

10.2.2 Choosing Among Alternative Reduction Schemata 119

10.2.3 Computing Planning Orderings 120

10.2.4 Relaxation of Some Restrictions on Multiple Goal Planning 120

10.2.5 Suboptimal Multiple Goal Plans 121

10.2.6 Preprocessing Search Spaces 122

A Action Templates and Reduction Schemata for the Blocks-World Domain **123**

B NP-Completeness of the Multiple-Goal Plan Existence Problem **129**

C Branch-and-Bound Search **131**

List of Tables

3.1 A List of Actions in the Painting Example 18

4.1 "fetch" example 41

8.1 Actions In An Example 99

8.2 Table of Results 104

List of Figures

1.1 Overview 3

2.1 Sussman Anomaly 7

2.2 Deleted-Condition Conflict 9

3.1 An Example of A Schema 17

3.2 Reduction Schemata 19

3.3 A Deleted-Condition Conflict 21

3.4 Imposing an Ordering Constraint 22

3.5 A Different Ordering 22

3.6 Aother Ordering Constraint 23

3.7 Last Way of Taking Care of the Conflict 23

3.8 A Top Level Plan 24

3.9 A Second Level Plan 24

3.10 A Plan Containing Conflicts 25

3.11 A Final Plan 26

3.12 An overview of the results in Chapters 4 and 5. 27

3.13 A plan containing unresolvable conflicts. 27

3.14 A Plan Containing Conflicts 28

4.1 Resolving conflicts by reducing a plan 31

4.2 A plan with four actions 33

4.3 Resolving Conflicts by Reduction 39

4.4 Modifying Reduction Schemata (1) 46

4.5 Modifying Reduction Schemata (2) 47

4.6 An Example of Augmented Protection Interval 50
4.7 A plan is $AU_Q(\Delta)$-unlinearizable. 51
4.8 Strongly-Unresolvable Conflicts 56
4.9 An Example of White Knight 57
4.10 A Plan Is Unlinearizable w.r.t. Other Constraints 59

5.1 A Plan Containing Conflicts 61

6.1 Action That Are Not Mergeable 73
6.2 A blocks-world Example 74
6.3 Machining Holes 76

7.1 Merging Plans in Different Ways 80
7.2 State Space 84
7.3 Experimental Results 90
7.4 Partial Ordering Restriction 91

8.1 More Than One Plan For Each Hole 93
8.2 Search Space 94
8.3 Example State Space 100
8.4 Experimental Results 105

9.1 An example search space 109
9.2 A search space with threads 109
9.3 Preprocessed search space 110
9.4 An AND/OR Tree 114

Chapter 1

Introduction

Planning is concerned with the problem of composing a course of actions that can transform the world from some given initial situation to some desirable goal situation. One of the differences between planning and other kinds of problem solving activities is that planning is often guided by goals and subgoals, and goals and subgoals usually interact. Of the various types of interactions, some are harmful, in which case a step for solving one goal may preclude another step for solving another goal. Other interactions may be useful, and one wants to take advantage of them. The task of detecting and handling the interactions has been a major concern for researchers in artificial intelligence planning.

Several approaches have been proposed for the problem of handling goal and subgoal interactions. Domain-independent planners aim at detecting and handling goal interactions based on the structure of the planning knowledge. They are generally applicable to many domains, but unfortunately, are not very efficient. Domain-dependent planners exploit powerful domain-dependent heuristics for detecting and handling goal interactions. They are generally more efficient than their domain-independent counterparts. However, because of their domain-dependent nature, it is hard to apply their planning technique to other domains. A research goal is to improve the efficiency of domain-independent planning and broaden the applicability of domain-dependent planning.

This thesis is divided into two parts. In Part I, methods are discussed for preprocessing the planning knowledge to obtain information about possible goal interactions. This information can be extracted before planning process starts, and used for guiding the problem solving activity during planning. In particular, these preprocessing methods are applied to hierarchical planning, in which a plan is generated by first buiding a high level plan and then refining it into more detail. The information extracted during preprocessing can be used in deciding when backtracking is needed, as well as how orderings should be imposed in a plan for taken care of goal interactions. In Part II, an approach is presented for improving planning efficiency by imposing restrictions on the kinds of goal interactions which are allowed to occur. By imposing restrictions on the goal interactions, it is possible to design efficient planning methods. This approach can be considered as an intermediate approach to planning as compared to both domain-independent and domain-dependent planning methods. To make this approach useful, the restrictions upon goal interactions should satisfy the following properties:

1. the restrictions are stateable in a clear and precise way (rather that simply referring to general knowledge about the characteristics of a particular domain of application);
2. the resulting classes of planning problems are large enough to be useful and interesting;
3. the classes of problems allowed are "well-behaved" enough that planning may be done with a reasonable degree of efficiency.

This approach is called *limited-interaction* planning.

1.1 Overview

A reader's guide to the thesis is provided as a plan in Fig. 1.1. In Chapter 2, past planning work will be briefly reviewed. Both domain-independent and domain-dependent planning methods will be discussed in the review. In particular, the limitation of both approaches to planning will be pointed out, and the motivation to the thesis is introduced.

Chapters 3, 4 and 5 will focus on preprocessing in hierarchical planning. Hierarchical planning is one of the prominent methods of abstraction in planning. Given a plan

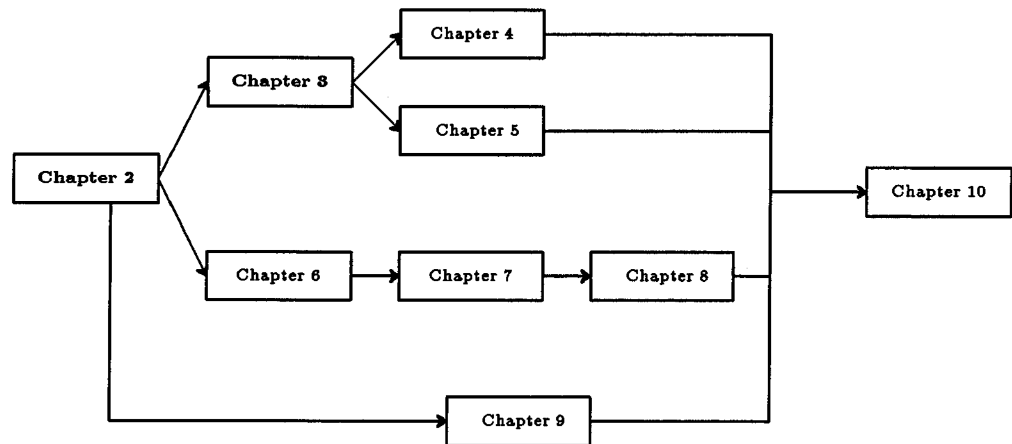


Figure 1.1: A reader's guide to the thesis as a plan.

composed of high level actions to be carried out, a hierarchical planner will find ways to expand each action into a set of lower level subactions according to some predefined action reduction schemata. At each step of expansion, critics will be used to find out and take care of the interactions between actions in the plan. This will be reviewed in Chapter 3.

It is sometimes possible for a hierarchical planner to encounter a situation where a plan contains a set of deleted-condition conflicts that are unresolvable. At this point, it is not always the correct behavior to backtrack, since a set of unresolvable conflicts in a plan may turn out to be resolvable after further reduction of the plan. In Chapter 4, a set of restrictions will be provided. These restrictions define the relationship between a non-primitive action and its set of sub-actions. When the restrictions are satisfied, it is possible to determine whether further reduction of a plan will make it possible to resolve previous unresolvable conflicts in it.

In Chapter 5, a property of hierarchical system, the *ordering-induced independence* property will be defined. Ordering-induced independence property is satisfied by a pair of actions (a, b), if a precedence ordering from a to b will not be undone because of interactions between the subactions of a and b. Again restrictions will be presented to restrict the way action reduction schemata are defined. When these restrictions are satisfied by a pair of actions, the ordering-induced independence property can be guaranteed.

In Chapter 6, a multiple goal planning problem, the least-cost multiple goal planning problem, will be defined formally, and the types of allowable goal interactions will be discussed. It will be shown in Chapter 7 that when one plan is available for each goal, the multiple goal planning problem is NP-hard to solve. However, after imposing restrictions on the interactions between the goals, polynomial time algorithm is available for solving the problem.

When more than one goal is available for each goal, the multiple goal planning problem will again be NP-hard, even with the restrictions of Chapter 7. This is shown in Chapter 8. In the same chapter, a heuristic algorithm will be presented along with empirical results.

Chapters 7 and 8 concentrate on the least-cost multiple goal planning problem. In Chapter 9, a method for preprocessing search spaces for branch-and-bound search will be discussed. This preprocessing method can be used for planning a individual goal. Finally, a summary of the thesis will be given in Chapter 10, along with a discussion of future work.

Chapter 2

Background

This chapter reviews previous work on planning. The review is divided into three parts: domain-independent planning systems, domain-dependent planners, and finally a discussion of the limitations of both along with motivation to the thesis.

2.1 Goal Interactions in Planning

Given a set of actions, a planning problem in A.I. is to compose a course of actions that can transform the world from some initial state to a desired goal state. The course of actions, called a *plan*, is a set of actions along with certain constraints on them. One constraint imposes orderings among the actions in a plan. For example, to drive a car from location A to location B, a plan may consist of a sequence of two actions, namely, starting the car, and then driving it from location A to location B. Notice that ordering is important in this plan; one can only drive the car *after* starting it up, but not the other way around.

Planning is useful for a wide range of applications, including automated manufacturing, electronic circuit design, robot control problems, etc, and is also closely related to some

other A.I. problems such as natural language understanding. To make it possible for planning to be automated by a computer program, it is important to address a number of research issues. These issues include the representation of the world, the representation of and reasoning about actions and the effects of actions upon the world, and the control of search processes.

Given a goal state to be achieved, a planner has to decide, among other things, which actions to choose for a plan, what orderings to impose upon the actions and which objects to use. This involves search in a space of possible plans. Usually, there are a large number of possibilities for a planner to choose from, and most of them cannot be used to achieve the goal. For example, the number of possible sequences of actions is usually exponential with the number of available actions. To ease the computational burden, planning systems usually decompose a complicated goal into two or more subgoals to solve. The reason for this is that decomposition tends to divide the exponent of an exponential problem, thus drastically reducing the total problem-solving effort. Korf [20], for example, has demonstrated that if the subgoals are independent, then solving each one in turn will divide both the base and the exponent of the complexity function by the number of subgoals.

The major assumption of the above analysis is that the goals are independent. This condition does not really hold for most planning problems. Instead, goals may interact or conflict with each other. One example of this is the "Sussman anomaly," in which solving one goal undoes the independently derived solution to the other. As illustrated in Fig. 2.1, suppose we have three blocks A, B and C, and we want to achieve the conjunctive goal

$$\mathrm{On}(A,B) \wedge \mathrm{On}(B,C)$$

given the initial situation

$$\mathrm{On}(C,A) \wedge \mathrm{On}(A,\mathrm{Table}) \wedge \mathrm{On}(B,\mathrm{Table}).$$

If only one block can be moved at a time, achieving either of the two goals $\mathrm{On}(B,C)$ and $\mathrm{On}(A,B)$ makes the other goal impossible to achieve without violating the one already achieved.

The goal interaction in the above example can be referred to as the *deleted-condition conflict.* It occurs when one way of achieving one of the goals deletes some of the conditions

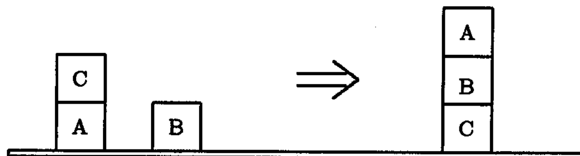


Figure 2.1: Sussman anomaly.

necessary for the achievement of the other goals. In this case, achieving On(B, C) by moving B to the top of C in the initial situation makes it impossible to move A any further, since now both B and C are above A. This makes it impossible to achieve On(A, B) without undo the actions for achieving On(B, C).

Deleted-condition conflict is only one kind of goal interaction that occur in planning, and many other types of goal interactions exist. Consider planning for the minimum cost plans in which every action is associated with a cost. Sometimes it is possible to reduce the total cost of a plan by performing several actions together. For example, suppose someone wants to buy both bread and milk. If both bread and milk are bought together instead of on separate trips to a store, then one can save an additional trip. In this case, the actions are said to be interacting with each other through *action-merging interactions.* This type of goal interaction can be considered helpful for planning, and should be taken advantage of. In [46], Wilensky presented a fairly complete classification of the types of goal interactions that occur in common sense domains.

One of the major difficulties in dealing with goal interactions is that the interactions do not usually exist in isolation; taking care of an interaction in a plan in a certain way may affect some other interactions in the same plan. For example, removing a deleted-condition conflict may create new conflicts, or make other conflicts more difficult to handle. Moreover, taking care of one type of interaction may affect interactions of some other types in the same plan. For example, one way to remove a deleted-condition conflict in a plan is to find a new action which can restore the deleted condition, and insert it between the

two actions in conflict. However, this new action may in turn create conflicts for resources with some existing actions, if they share the same scarce resources. Thus, taking care of the deleted-condition conflict affects conflicts which involve resources.

A number of approaches have been suggested for dealing with interactions in planning. They can be generally classified as domain-independent and domain-dependent. In the following, both approaches are reviewed, followed by a discussion of the motivation to the thesis.

2.2 Domain-Independent Planning

Domain-independent planners are based on the structure of the planning knowledge, without resort to domain-specific heuristics. They have the advantage of being applicable to many domains and provide planning techniques capable of dealing with many types of goal interactions. Domain-independent planners can be classified as either *linear* or *nonlinear*, according to how goal interactions are treated.

Many of the earlier domain-independent planners depend on the condition of *linearity*. The condition of *linearity* in a planning problem is satisfied if the subgoals are all independent and can be achieved sequentially in any arbitrary order. Linear planners start by generating plans for the subgoals as if the planning problem were linear. They apply actions one at a time, and the resultant plans are linear sequences of actions. STRIPS [11] and HACKER [41] are examples of linear planners.

The linearity assumption is appropriate in cases where there is no *a priori* reason to order one operator ahead of another. This can be considered as a domain-independent heuristic for planning. However, the problem with this heuristic is that it may lead a planner to make a premature commitment to an incorrect ordering of interacting subgoals—and much backtracking and additional computationally expensive search may be necessary to achieve the final result. For example, consider the problem of painting a ceiling and a ladder(based on [38]). The goal can be formulated as the conjunction of two subgoals:

Achieve(Painted(Ceiling)) and Achieve(Painted(Ladder)).

A linear planner solves this problem by solving either one of the subgoals first. Suppose it chooses to paint ladder first. This would involve getting the paint and apply paint to

the ladder. However, after painting the ladder it is no longer possible to paint the ceiling, since painting the ceiling requires that the ladder be available to be climbed. At this point, it is necessary to backtrack to try the other ordering of the goals, namely, painting the ceiling before painting the ladder.

Therefore, to use a linear planner in domains where subgoals or goals interact strongly, it is necessary to add ways to detect and resolve the conflicts. As an example, STRIPS [11] did linear planning for compound goals which are conjuncts of component goals. Even with these fixes, linear planners are not able to find the most efficient solutions in cases where it is necessary to interleave the actions for achieving different goals, and, in some of the extreme cases, they are not able to find a solution at all. The Sussman anomaly is an example for which a linear planner is not able to find the most efficient solution.

Another domain-independent heuristic is to make planning decisions only when they are needed. Thus, if there is no conflict between two actions, then they should be left in unordered. This results in the *least-commitment strategy.* Consider the plan in Fig. 2.2 where an action a deletes a precondition p of another action b. One way to handle this conflict is to introduce an ordering so that a occurs after b. An alternative way to handle this conflict is to order a *before* c, which asserts the condition p for b. The plan thus developed is a partially ordered set of actions, and for this reason, planners using the least-commitment strategy are often referred to as nonlinear planners. Most of the well-known planning systems (for example, [38, 42, 29, 45, 7, 47, 40]), fall into this category.

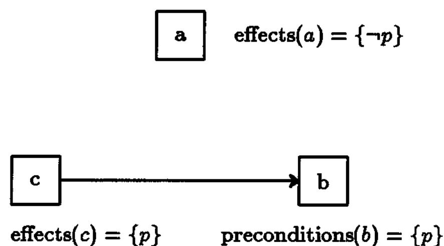


Figure 2.2: A deleted-condition conflict in a plan.

Although domain-independent nonlinear planners aim at reducing the unnecessary backtracking experienced by their linear counterparts, there are still serious computational problems involved. One problem is that by delaying commitment to ordering constraints in a plan makes, it is more difficult to verify the correctness of a plan. Plan verification is needed to check if all of the intended effects of the actions in a plan are indeed true at places where they are needed. This process is an important step in the detection of interactions in a plan. However, the problem of plan verification has been shown to be NP-hard in most practically interesting cases ([10, 7]). Dean and Boddy [10] proposed to overcome the problem by using verification algorithms that are incomplete in nature for a number of problems. But the general computational problem still remains as a serious limitation to nonlinear planning.

Another problem with nonlinear planners is that it is difficult to control search in an efficient way. At all times, a nonlinear planner has a *partial* plan to complete. A plan is partial either because more orderings are needed, because some of the variables in the plan are to be instantiated, or because more actions need to be inserted. Thus, planning corresponds to a search in a space of partial plans, where the problem solving operators correspond to plan modification operators such as one that imposes an ordering. This search space could be huge, and good heuristics are needed to guide the search. However, most planners correspond to only depth-first searches in this state space. Thus, more work needs to be done in order to find good heuristics for domain-independent nonlinear planners.

2.3 Domain-Dependent Planning

One way to lessen the computational burden of planning is to make use of domain-dependent knowledge to detect and handle goal interactions. In the MACHINIST system (for planning of machining operations on a metal part), Hayes [15] uses a set of production rules to detect interactions between machining features. These rules also suggest ways handle the interactions in a correct manner. For example, suppose a hole and a flat surface are to be made, and the hole and the flat surface intersect each other at an angle other than 90°. If the flat surface is made first, the then drill bit will slip on the flat surface

when creating the hole, making the hole placement inaccurate. Therefore, a rule tells the program to make the hole before making the surface. Using these rules, the MACHINIST planner can perform more efficiently than some of the experienced machinists.

Domain-dependent planners are widely used in practice. Other examples include military command and control planning [3, 14, 5], route planning [21, 13], autonomous vehicle navigation [4, 24], and automated manufacturing [6, 9, 31]. Although these planners all work efficiently in their own application domains, their domain-dependent nature makes it difficult to apply them to other domains.

2.4 Motivation

As discussed above, the treatment of goal interactions in previous planning research still presents major problems. On the one hand, domain-independent planners deal with interactions using domain-independent planning knowledge, and are applicable to many domains. But they are generally not very efficient. On the other hand, domain-dependent planners can be more efficient in their own domain of application, but their range of applicability is too narrow. New methods are needed that are efficient in handling goal interactions and are also general enough to be applicable to many domains.

In this thesis, two methods are presented for the improvement of planning efficiency. First, the representation of the actions of a planner provides information about the interactions between actions. This information can be extracted *before* the planning process starts, and can be used in various ways during planning. In the first part of the thesis, this issue is explored in the context of hierarchical planning[1]. Planning can be considered as a state space search, and methods are needed to tell if a state is a *dead end*, where backtracking is needed. A dead end in hierarchical planning corresponds to a plan containing conflicts that are unresolvable. The earlier such dead ends are detected, the more efficient the planning process can be. In addition, a search is more efficient if it is possible to tell which path to explore first given more than one alternative path to choose from. Such decision is needed in planning when ordering needs to be imposed among the actions in a plan that conflict with each other. Chapter 4 presents methods that enable a planner

[1] Hierarchical planning will be discussed in detail in the next chapter.

to recognize unresolvable conflicts early in planning by preprocessing a set of actions. In Chapter 5, a property of the actions, which can be checked *a priori*, is established that enables a planner to make intelligent decisions when facing multiple ways for taking care of a conflict.

A second technique for improving planning efficiency is presented in Part II. As discussed in the previous sections, one source of computational difficulty of nonlinear planners lies in their attempt to handle a comprehensive set of interactions efficiently. A more realistic approach may be to attack specific classes of planning problems by allowing only certain kinds of goal interactions. If the domain-dependent and domain-independent planners are considered as on the extremes of a spectrum, respectively, then this approach can be thought of as one which falls in the middle of the spectrum. For this reason, it will be referred to as *limited-interaction planning.* In the second part of the thesis, this approach is applied to the multiple goal plan optimization problem, which selects the least cost plan for achieving more than one goal. With this approach, instead of generating a plan for a single conjoined goal, plans are generated for each goal concurrently, and then an "optimization" step is performed to merge the resultant plans into a single global plan. During this optimization step, allowable interactions between the goals are taken care of. Finding the best plan will be shown to be NP-hard in general, but it is possible to design a set of restrictions that limit the allowable interactions while permitting the process to be performed efficiently. For the limited-interaction planning approach to be useful, the allowable set of interactions must result in a class of planning problems that is broad enough to be useful and interesting—and yet "well-behaved" enough that reasonably efficient planning procedures can be developed.

The above approach to multiple goal planning relies on the ability to efficiently generating plans for each individual goal. A class of planning problems, such as process planning and robot route planning, can be solved using branch-and-bound search algorithms. Chapter 9 presents a method for extracting the control information of the branch-and-bound search, and for making use the information during the actual problem solving process. Specifically, a structure for "remembering" the control information, called the *threaded decision graph*, is used to replace the ordered set of incomplete paths for branch-and-bound search. With the graph, no manipulation of the set is done, and therefore, efficiency is improved.

Part I

Improving the Efficiency of Hierarchical Planning

Chapter 3

Hierarchical Planning—Definitions

Hierarchical planning using action reduction is one of the most widely used planning methods. Given a set of high level actions to be carried out, this method will find ways for reducing each action into subactions according to a predefined set of action reduction schemata. It then resolves possible conflicts among the subactions in the plan using the least-commitment strategy. The process is repeated until all of the actions in the plan are primitive, and all of the interactions between the actions are removed. The advantage of this planning method is that a hierarchical planner works on a small but important set of interactions first, before it sets out to handle the rest which are considered as mere details. NOAH[38] and NONLIN[42] are two of the well known hierarchical planners.

Historically, the term "hierarchical planning" has been used for mainly two types of planning, and the kind of hierarchical planning in this thesis only refers to one of them. Another kind of hierarchical planning is used in ABSTRIPS ([35]). The definition given in this chapter cannot include all possible kinds of nonlinear planners which decompose higher level actions into low level ones. However, one of our goals is to include as many

kinds of hierarchical planning as possible.

There are two parts to hierarchical planning with action reduction. The first is the representation of actions and their reductions, and the second is the way problem solving is done using these representations.

3.1 Representation

The planning knowledge of a hierarchical planning system consists of two parts. The first part is a set of action templates Λ. Each action template $\mathbf{a} \in \Lambda$ is represented in terms of preconditions and effects. Specifically, each action template **a** has a set of preconditions preconditions(**a**), and a set of effects effects(**a**), where each set consists of literals in first-order predicate logic. Thus, for example, one can choose to represent the action template for moving a block x from the top of another block y to the table in the blocks-world domain as

put-block-on-table(x, y)

comment: *move x from top of y to the table.*

preconditions=$\{\text{Block}(x), \text{Block}(y), \text{Cleartop}(x), \text{On}(x, y)\}$

effects=$\{\text{Ontable}(x), \text{Cleartop}(y), \neg\text{On}(x, y)\}$

An action template in Λ can be *instantiated* by replacing some of the variables in its preconditions or effects by terms. The replacement can be represented by a set of ordered pairs , where the first element of each pair is a variable and the second element is a term. Each such set is called a *substitution.* If **a** is an element of Λ and

$$\beta = \{\langle v_1, t_1\rangle, \langle v_2, t_2\rangle, \ldots, \langle v_n, t_n\rangle\}$$

is a subtitution, then $\mathbf{a}\beta$ is **a** with every variable v_i that appears in its preconditions and effects replaced by t_i. $a = \mathbf{a}\beta$ is called an *instance* of **a**, and is also referred to as an *action.* In order to make a clear distinction between an action template and an action, the former is denoted in boldface, while the latter is not. If a is an instance of an action template

$\mathbf{a}$, then the latter is called a *template* of the former, and template(a) = $\mathbf{a}$. It is clear that each action has a unique template.

The second component of the planning knowledge of a hierarchical planner is a set of *action reduction schemata* Φ. A reduction schema $R \in \Phi$ is a function which, when applied to an action template in Λ, returns a partially ordered set of actions a_i. R is not necessarily applicable to every action template in Λ, and an action template can have more than one reduction schema applicable to it. The set of action reduction schemata applicable to $\mathbf{a}$ is denoted by $\alpha(\mathbf{a})$. $\mathbf{a}$ is *primitive* if $\alpha(\mathbf{a}) = \emptyset$, otherwise it is *non-primitive.* Intuitively, a primitive action template is one which cannot be decomposed further into more detailed steps, while a non-primitive one can.

If R is applicable to $\mathbf{a}$, then $R(\mathbf{a}) = \langle A, E, C \rangle$, where A is a set of actions, E defines a partial ordering among the elements of A, and C is a set of conditions along with specifications of where they must hold. $R(\mathbf{a})$ is called a *reduction* of $\mathbf{a}$. Each element of C is called a *protection interval* [8], which is a triple $\langle p, a_1, a_2 \rangle$, where p is a condition, and a_1, a_2 are actions in A such that

1. $p \in$ effects(a_1),

2. $p \in$ preconditions(a_2), and

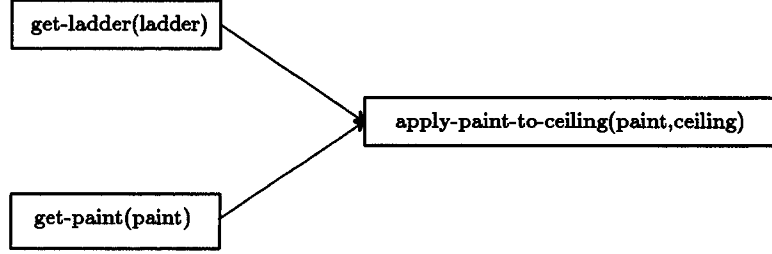
3. p is expected to be true after a_1 and before a_2.

$\mathbf{a}_1$ is called a *provider* of p for $\mathbf{a}_2$.

Let $A(R(\mathbf{a}))$ be the set of actions in $R(\mathbf{a})$. In the following discussion, $a \prec b$ means that a is constrained to occur before b. The actions in $A(R(\mathbf{a}))$ have to obey the following restrictions:

1. $\forall e \in$ effects($\mathbf{a}$), $\exists a_i \in A(R(a))$ such that $e \in$ effects(a_i) and $\forall b \in A(R(\mathbf{a}))$, if $\neg e \in$ effects(b) then $b \prec a_i$. That is, every effect of an action template is asserted by at least one of the actions in its reduction.

2. $\forall p \in$ preconditions($\mathbf{a}$), $\exists a_i \in A(R(\mathbf{a}))$ such that $p \in$ preconditions(a_i), and $\forall b \in A(R(\mathbf{a}))$, if $p \in$ effects(b) then $a_i \prec b$. That is, every precondition of an action template is also a precondition of at least one of the actions in its reduction, and futher this precondition is not the effect of some earlier subaction.



$\langle$**Have(*ladder*), get-ladder(*ladder*),apply-paint-to-ceiling(*paint, ceiling*)**$\rangle$

$\langle$**Have(*paint*), get-paint(*paint*), apply-paint-to-ceiling(*paint, ceiling*)**$\rangle$

Figure 3.1: A reduction schema for painting a ceiling.

3. $\forall a_i \in A(R(\mathbf{a})), \forall p \in$ preconditions$(a_i), \forall b \in A(R(\mathbf{a}))$ such that $a_i \not\prec b$, if $\neg p \in$ effects(b) then $\exists c \in A(R(a))$ such that $c \prec a_i$ and $b \prec c$, and $p \in$ effects(c). In other words, a reduction is a miniature plan free of any conflicts.

The definition of reduction schemata above is intended to capture the formal aspects of action or goal expansions in a number of systems. In particular, a reduction schema R corresponds to a "soup code" in NOAH [38], an "opschema" or an "actschema" in NONLIN [42], and a "plot" in SIPE [47]. Also, we made no distinction between a goal and an action in our formalization; all the goals are represented as action templates in Λ which can be reduced to a special action **no-op**, a primitive action which means the action can be achieved by doing nothing.

Let $a = \mathbf{a}\beta$ be an instance of $\mathbf{a} \in \Lambda$. We extend the definition of action reduction schemata to instances of action templates, as follows: $\forall R \in \alpha(\mathbf{a})$, $R(a) = R(\mathbf{a})\beta$, where $R(\mathbf{a})\beta$ is the result of applying the substitution β to all the actions in $A(R(\mathbf{a}))$, and to all the action occurrences in $C(R(\mathbf{a}))$. If R is applicable to a, then $R(a)$ is called a *reduction* of a, and each action in $R(a)$ is called a *subaction* of a. Intuitively, A reduction for an instance a of $\mathbf{a}$ consists of a partially ordered set of actions, each of which is an instance of an action template in $A(R(\mathbf{a}))$. Extended in this way, it is clear that the following

action	preconditions	effects
paint-ceiling($ceiling$)	$\emptyset$	{Painted($ceiling$)}
get-ladder($ladder$)	$\emptyset$	{Have($ladder$)}
get-paint($paint$)	$\emptyset$	{Have($paint$)}
apply-paint-to-ceiling($paint, ceiling$)	{Have($ladder$), Have($paint$), Climbable($ladder$)}	{Painted($ceiling$), ...}

Table 3.1: Actions in the painting example with their preconditions and effects.

properties hold: Let $a = \mathbf{a}\beta$. Then

1. $\alpha(a) = \alpha(\mathbf{a})$, and
2. a is primitive iff $\mathbf{a}$ is.

Fig. 3.1 describes an action reduction schema for painting a ceiling. A set of protection intervals is also given in the figure. The preconditions and effects of each action are given in Table 3.1.

Let Λ be the set of actions and Φ be the set of action reduction schemata in a planning system's planning knowledge. The following definition characterizes what is meant by "descendents" of an action template.

Definition 3.1 *Let* $\mathbf{a} \in \Lambda$ *be an action template.* $\mathrm{TC}(\mathbf{a})$ *defined as follows:*

1. $\mathbf{a} \in \mathrm{TC}(\mathbf{a})$,

2. If $\mathbf{a}' \in \mathrm{TC}(\mathbf{a})$, *then* $\forall R \in \alpha(\mathbf{a}')$ *if* $a'' \in A(R(\mathbf{a}'))$, *then* $\mathrm{template}(a'') \in \mathrm{TC}(\mathbf{a})$.

As an example of TC(a), consider the set of action reduction schemata in Fig. 3.2. For the action template $\mathbf{a}$ in the figure,

$$\mathrm{TC}(\mathbf{a}) = \{\mathbf{a}, \mathbf{a_1}, \mathbf{a_2}, \mathbf{a_3}, \mathbf{a_4}\}.$$

For a given Λ, suppose each action template has a maximum of k action reduction schemata applicable to it. Then the relationship between actions and subactions can be represented as a directed graph, with $k \times |\Lambda|^2$ edges. For any given action template $\mathbf{a}$, the computation of TC($\mathbf{a}$) can be implemented as a depth first search in this graph. Thus, the worst case time complexity for computing TC for one action template is $O(k \times |\Lambda|^2)$.

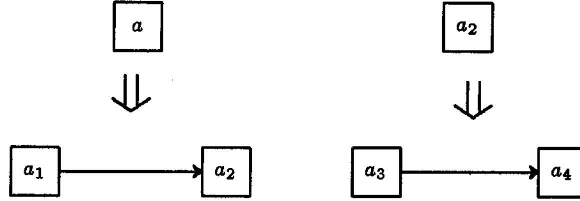


Figure 3.2: A set of plan reduction schemata

3.2 Planning with Action Reductions

A plan is defined as a partially ordered set of actions. Let P be a plan, then $P = \langle A_P, E_P, C_P \rangle$, where A_P is a set of actions, E_P defines a partial ordering among the actions in A_P, and C_P is a set of protection intervals. Each action in A_P is an instance of some action template in Λ.

Let a be an action in a plan P, and R be a reduction schema applicable to a. Then $R(P)$ is the plan which results when a is replaced by $R(a)$ in P. Let $\delta = \langle p, a, b \rangle$ be a protection interval in P. δ is not associated with $R(P)$, instead $R(P)$ will have one or more protection intervals derived from δ which involve subactions $a_i \in A(R(a))$. In particular, let a_i be a subaction of a such that

1. $p \in$ effects(a_i), and

2. $\forall a_j \not\prec a_i$, $\neg p \notin$ effects(a_j).

By the definition of R, one or more such a_i always exists. Then $\langle p, a_i, b\rangle$ is a protection interval associated with $R(P)$.

Similarly, let b be an action in a plan P, and R' be a reduction schema applicable to b. Let $R'(P)$ be the plan which results when b is replaced by $R'(b)$ in P. Suppose $\langle p,\ a,\ b\rangle$ is a protection interval in P. Then $R'(P)$ is associated with one or more protection intervals involving the subactions of b. Let b_i be a subaction in $A(R'(b))$, satisfying the following conditions:

1. $p \in$ preconditions(b_i),

2. $\forall b_j \in A(R(b))$, if $p \in$ effects(b_j) then $b_i \prec b_j$.

Then $\langle p, a, b_i\rangle$ is a protection interval associated with $R'(P)$.

Let P be a plan, and $R_1, R_2, \ldots, R_n$ be reduction schemata. Then

$$Q(P) = R_1(R_2(\ldots(R_n(P))\ldots)$$

is a *composite reduction* of P. Furthermore, if Δ is a set of protection intervals associated with P, then $Q(\Delta)$ is the corresponding set of protection intervals associated with $Q(P)$.

A hierarchical planner starts planning for a given set of goals by finding appropriate actions to achieve them. If $G(X)$ is a goal, then the action chosen to achieve this goal should have $G(x)$ as one of its effects. These actions form a plan at the highest level. The subsequent problem solving process can be described in the following steps:

1. Choose a non-primitive action, and replace it by one of its reductions. Let the new plan be P.

2. Find out the set of interactions among the actions in P, and suggest ways to handle them.

3. If all the actions in P are primitive then terminate planning. Else go to step 1.

After step 1 is done, certain new interactions may appear. Depending on the particular domain of application, interactions can be of different types. In this part of the thesis, the

only type of interaction considered is *deleted-condition conflict* between a pair of actions. This type of interaction occurs when an action in a plan deletes a precondition or an intended effect of some other action. More formally, let P be a plan, and $\langle p, a, a' \rangle$ be a protection interval in P. Suppose there is some action b in P such that $\neg p \in$ effects(b). If $b \not\prec a$ and $a' \not\prec b$, then a deleted-condition conflict occurs in the plan P (see Fig. 3.3). This conflict can be characterized according to whether $\neg p$ is also an intended effect of b:

1. both $\langle p, a, a' \rangle$ and $(\langle \neg p, b, b' \rangle$ are protected intervals in P, so the conflict is denoted by the pair $(\langle p, a, a' \rangle, \langle \neg p, b, b' \rangle)$. In this case, one way to remove the conflict is to impose a time ordering such that a' occurs before b (Fig. 3.4), or b' occurs before a (Fig. 3.5).

2. If $\neg p$ is not needed by any action in P, then it is not a protected condition, so the conflict can be denoted by the pair $(\langle p, a, a' \rangle, \langle \neg p, b \rangle)$. In this case, one way to remove the conflict is to order b before a (Fig. 3.6) or after a' (Fig. 3.7).

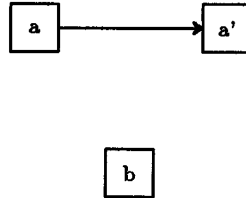


Figure 3.3: A deleted-condition conflict.

Consider the following example adopted from [38]. The goals are to paint a ladder and to paint a ceiling. A top level plan for the goal consists of a single conjunctive formula, as shown in Fig. 3.8. At the next level, this plan is broken down into two separate actions in parallel (Fig. 3.9). Since at this level of reduction, there is no interaction between the two parallel actions, the actions are further reduced into more detailed plans, consisting of several component steps (Fig. 3.10).

At this point, an interaction can be detected between the action apply-paint-to-ladder(Paint, Ladder) and the action apply-paint-to-ceiling(Paint, Ceiling), because if

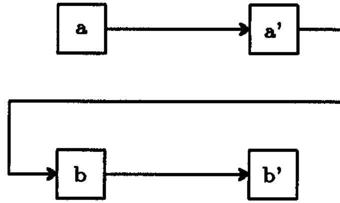


Figure 3.4: Romoving the conflict by introducing an ordering.

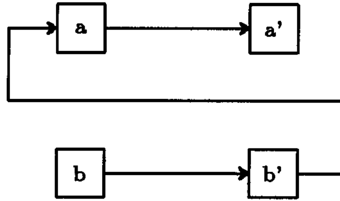


Figure 3.5: Removing the conflict by introducing an ordering.

the ladder is painted first, the precondition Climbable(Ladder) of the action apply-paint-to-ceiling(Paint, Ceiling) will be deleted. One way to handle nteraction is to order apply-paint-to-ladder(Paint, Ladder) after the action apply-paint-to-ceiling(Paint, Ceiling), resulting in the plan in Fig. 3.11. At this point, every action in the plan is primitive, and no conflicts occur. Thus, planning can be terminated.

3.3 Overview

A hierarchical planning system achieves its efficiency by taking care of more important interactions first, and worrying about detailed ones later. This technique is made possible by representing the domain knowledge hierarchically in the form of action reduction

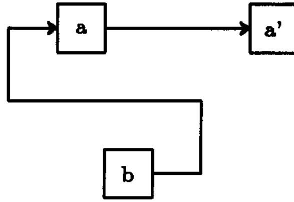


Figure 3.6: Removing the conflict by introducing an ordering.

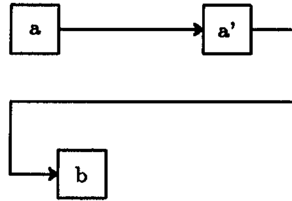


Figure 3.7: Removing the conflict by introducing an ordering.

schemata. The efficiency of hierarchical planning depends upon how the action reduction schemata are defined. If one is not careful in the definition, one can lose the efficiency of hierarchical planning. In the next two chapters, we will demonstrate this point in terms of two important properties of hierarchical planning systems.

First, if a planner faces conflicts in a plan that are impossible to resolve, in general it has to try to reduce its actions further in order to resolve the conflicts by interleaving the subactions. This greatly reduces the efficiency of a planner, since it has to search the an extra portion of the planner's search space, even if there is strong indication that no solution exists in that space. In Chapter 4, we show that a class of domains exists, in which the action reduction schemata can be defined so that unresolvable conflicts in a plan cannot be resolved in any reduction of the plan.

Second, at each step of planning, a hierarchical planning system has to constantly

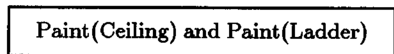


Figure 3.8: A top level plan containing a single node.

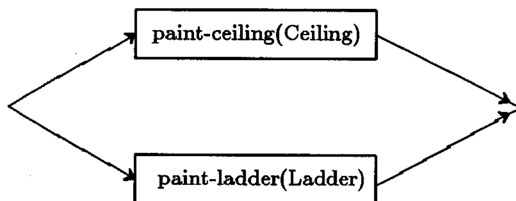


Figure 3.9: A plan at the second level of reduction.

make decisions on which ordering among a set of alternative ones to impose on a plan. Chapter 5 shows that it is often possible to extract information about goal interactions beforehand, and make use of them to make better decisions for the orderings.

More specifically, we discuss two properties in the next two chapters. The *downward-unlinearizability* property is defined as follows. Suppose a plan cannot be linearized while preserving all of its protected conditions. Then this plan corresponds to a "dead-end" to the planner. Thus, whenever such a property is satisfied, backtracking from an unlinearizable plan will not lose any solution. The *ordering-induced independence property* is satisfied by two actions a and b, whenever an ordering $a \prec b$ in a plan cannot be undone later by the subactions of either a or b. Associated with each property is a set of sufficient restrictions. We show through a set of theorems that these restrictions are imposed upon the action reduction schemata, and can guarantee both the downward-unlinearizability

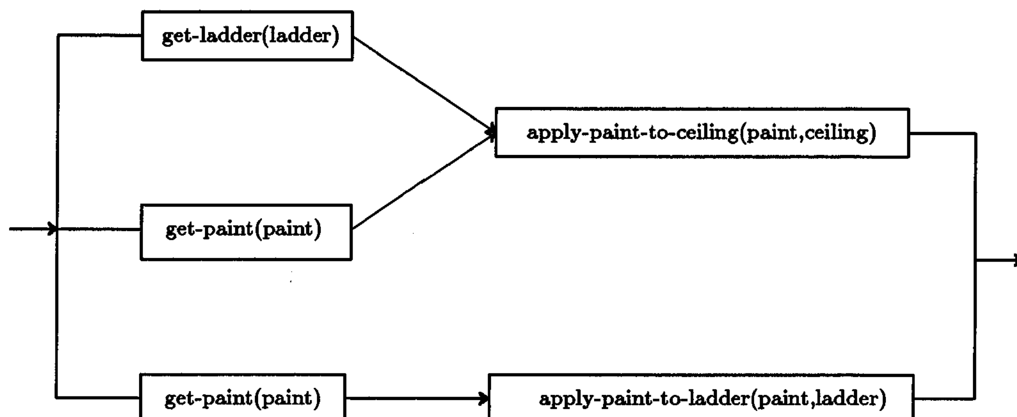


Figure 3.10: A plan containing interactions among its actions.

and the ordering-induced independence properties.

The results of Chapters 4 and 5 provide methods for *preprocessing* a given set of action reduction schemata Φ. Given Φ, one can use the syntactic restrictions to check if the downward-unlinearizability property is satisfied by the action reduction schemata in Φ. Also, one can check if any pair of action templates satisfies the ordering-induced independence property. If these properties are satisfied, then the actions and schemata are marked as so. The marked action templates and reduction schemata are then used for planning. This process is shown in Fig. 3.12, where Φ represents the set of given action reduction schemata, and Λ represents the set of given action templates.

We demonstrate how these two properties can be used during planning through the following examples. Suppose now that instead of painting the ceiling and the ladder, one wants to paint the ceiling and the floor. Assume the hierarchical planner has come up with a plan as shown in Fig. 3.13. Assume that if one paints the ceiling first, the paint will drip down and makes it impossible to paint the floor next. But if one paints the floor first, one cannot get in the room to paint the ceiling. That is, the action "apply-paint-to-ceiling" deletes a precondition for "apply-paint-to-floor", and the latter also deletes a precondition for the former. Therefore, the plan in Fig. 3.13 is unlinearizable. Suppose that the set

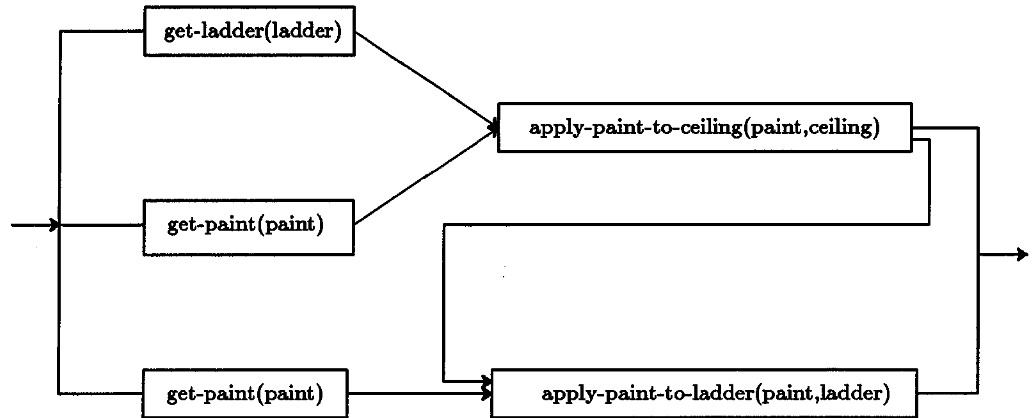


Figure 3.11: A plan with an interaction taken care of.

of action reduction schemata for the painting domain have been marked as satisfying the downward-unlinearizability property, it can *backtrack* from this plan, and try other ways to fulfill the goals. If no other way exists, the planner can announce failure. However, if the schemata does not satisfy the property, then the planner cannot rule out reducing the actions further as a means of resolving the conflicts. One such situation will be shown later in Fig. 4.1.

To see how a planner can make use of the ordering-induced independence property, consider a portion of a plan shown in Fig. 3.14. A conflict exists in this part of the plan involving two protection intervals $\langle p, a, b \rangle$ and $\langle \neg p, c, d \rangle$. To resolve this conflict, two orderings are possible: the planner can impose either $b \prec c$, or $d \prec a$. Suppose that through preprocessing, it is known that (b, c) satisfies the ordering-induced independence property, while (d, a) does not, then the ordering $b \prec c$ is prefered and tried first.

Equally important to preprocessing, these properties and restrictions also provide a standard over how the reduction schemata should be defined. For a given domain, there are usually several ways of defining the action reduction schemata. If one can define them in such a way that the action templates and the reduction schemata satisfy the restrictions, then planning can be done more efficiently.

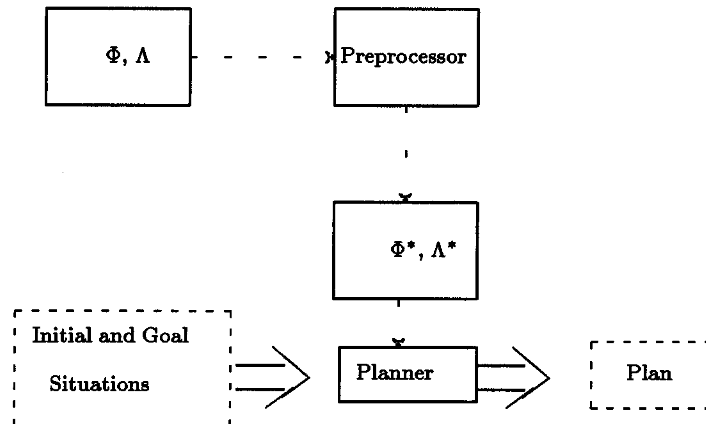


Figure 3.12: An overview of the results in Chapters 4 and 5.

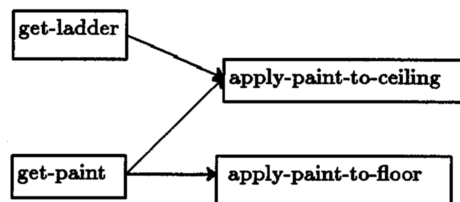


Figure 3.13: A plan containing unresolvable conflicts.

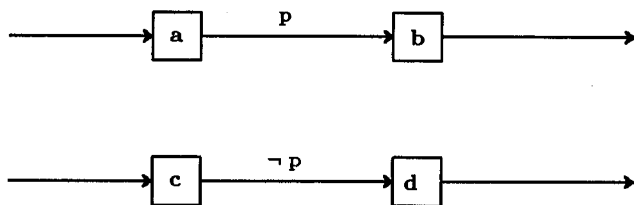


Figure 3.14: Part of a plan containing conflicts.

Chapter 4

The Treatment of Unresolvable Conflicts

4.1 Motivation

A hierarchical planner plans by repeatedly selecting and reducing the non-primitive actions, and applying critics to detect and resolve conflicts in a plan. Sometimes the critics may find conflicts in a plan that are impossible to resolve. A plan in this situation is said to have *unresolvable conflicts.* A number of planners such as NONLIN [42] or SIPE [47] will *backtrack* from such a plan to try other ways to achieve the goals, or announce failure if no alternative ways exist. However, this is not always the correct behavior, because there exist situations in which conflicts that are unresolvable in a plan may be resolvable in a composite reduction of the plan.

As an example of this, suppose a plan contains two higher-level actions a and b, with no constraints on their relative ordering (Fig. 4.1 (a)). Suppose that their effects and preconditions are

preconditions(a) = $\{x\}$, effects(a) = $\{u, \neg y\}$,

preconditions(b) = $\{y\}$, effects(b) = $\{w, \neg x\}$,

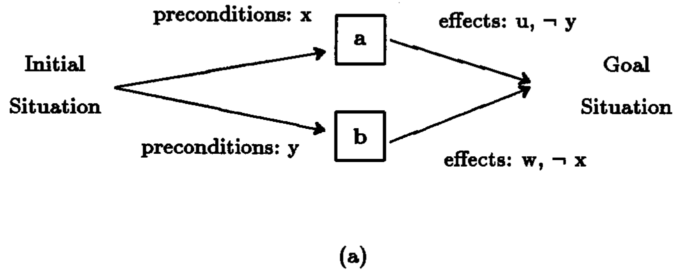
where the propositions u, w, x, y are all distinct. Then the following conflicts will occur: an effect of a deletes a preconditions of b, and a effect of b deletes a precondition of a. This kind of conflict is often called a "double cross," meaning that neither possible ordering of a and b will work. Unless a planning system can resolve this conflict by inserting additional actions between a and b to restore the needed preconditions, it will normally announce failure. However, suppose that

1. action a can be reduced to the subactions $a_1 \prec a_2$, with preconditions(a_1) = preconditions(a), effects(a_2) = effects(a), and $x \notin$ preconditions(a_2), and
2. b can be reduced to $b_1 \prec b_2$, with preconditions(b_1) = preconditions(b) and effects(b_2) = effects(b), and $y \notin$ preconditions(b_2).

Then the interaction can be resolved in the reduced plan by assigning orderings such that $a_1 \prec b_2$ and $b_1 \prec a_2$ (Fig. 4.1 (b)).

In the above example, a conflict which appears unresolvable at a higher level is in fact resolvable at a lower level. Therefore, a planning system should consider reducing a plan as a way to resolve unresolvable conflicts.

However, it is undesirable to do this for the following two reasons. First, an unresolvable conflict usually indicates that it is unlikely for a solution to be found, no matter how the solution is refined. It would be desirable for a planner to have the property that whenever a solution exists at the lower level of abstraction, the abstract version of the solution should also be a solution at the upper level. Tenenberg [43] called such a property the *upward-solution property* for ABSTRIPS type of hierarchical planning, in which a higher level action can be created by eliminating conditions of a low level action which are considered as less important. By imposing certain syntactic restrictions on how the ABSTRIPS abstraction hierarchy is defined, Tenenberg proved that the upward-solution property holds. One purpose of this chapter is to also ensure this property for the hierarchical planning systems with action decomposition. Second, the necessity for a planner to search through one more branch in its search space makes planning more inefficient. If one can ensure that a planner can backtrack whenever unresolvable conflicts occur, and



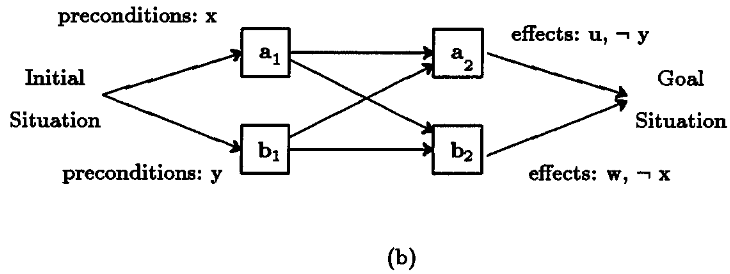


Figure 4.1: (a) A plan with unresolvable conflicts, (b) Resolving the conflict by reducing the plan in (a).

that this can be done without losing any possible solutions, planning efficiency can be improved. For certain domains, such a property can be enforced by imposing a set of syntactic restrictions on how the action reduction schemata should be defined. For others, such a property is not satisfied by all of the actions, but the syntactic restrictions could enable one to preprocess the set of action reduction schema so that all the actions which satisfy the restrictions can be identified *a priori.*

In the following sections, we will first define unlinearizability of a plan, and design restrictions to be imposed on reduction schemata so that unresolvable conflicts in a plan indicate that the conflicts cannot be resolved in any composite reduction of the plan.

4.2 Plan Unlinearizability

4.2.1 Definitions

A partially ordered plan has many possible *linearizations*, each being a totally ordered sequence of actions in the plan. If a and b are actions in a plan P and L is a linearization of P, then $a \prec b$ in L if $a \prec b$ in P. A plan can also have different instantiations of its variables. Any instantiated and linearized plan that is free of conflicts can be viewed as an instance of the original plan, and can be used to achieve the given goals. In the following, the word "linearization" of a plan refers to any "ground" linearization of the plan, where all the terms in a substitution β are constants.

Definition 4.1 *Let $\delta = \langle p, a, b \rangle$ be a protection interval in a plan P. δ is* violated *in the plan iff $\exists c \in A(P)$ such that $a \prec c$, $c \prec b$, and $\neg p \in \text{effects}(c)$.*

In general, a plan may contain several deleted-condition conflicts. For such a plan, we would like to know whether or not a linearization exists in which none of the protection intervals is violated.

Definition 4.2 *Let P be a plan and Δ be a set of protection intervals in P. P is said to be* Δ-linearizable *iff there exists a linearization L of P such that $\forall \delta \in \Delta$, δ is not violated in L. If L exists, it is said to be* Δ-consistent.

For example, for the plan in Fig. 4.2, the set of all protection intervals is

$$\Delta = \{\langle p, c, a \rangle, \langle w, a, d \rangle, \langle q, c, b \rangle, \langle \neg p, b, d \rangle\}.$$

For this plan, the linearization $c \prec a \prec b \prec d$ is Δ-consistent.

A linearization of a plan is Δ-*inconsistent* if it is not Δ-consistent. A plan P is Δ-*unlinearizable* iff every linearization of P is Δ-inconsistent. If Δ contains a set of protection intervals in a plan P, and if P Δ-unlinearizable, then P is said to contain "unresolvable conflicts." For example, the plan in Fig. 4.1 (a) contains unresolvable conflicts, since it is Δ-unlinearizable for

$$\Delta = \{\langle x, i, a \rangle, \langle y, i, b \rangle\},$$

where i represents the initial situation.

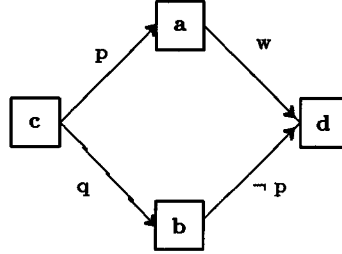


Figure 4.2: A plan with four actions, a, b, c and d, along with protection intervals in the plan.

4.2.2 Imposing Syntactic Restrictions

Let P be a plan, Δ be a set of protection intervals in P, and $Q(P)$ be a composite reduction of P. In this section, a set of restrictions will be imposed upon action reduction schemata to guarantee that $Q(P)$ is $Q(\Delta)$-unlinearizable if P is Δ-unlinearizable. To begin with, it is important to note that different actions are related to Δ in different ways, and different kinds of restrictions are needed. The following definitions aim at capturing these differences.

Let δ be a protection interval and a be an action in P.

Definition 4.3 *a is* e-relevant *for δ iff either*

1. $\delta = \langle p, a, b \rangle$ *for some p and b, or*

2. $\delta = \langle \neg p, c, d \rangle$*, where* $p \in \text{effects}(a)$*,* $a \not\prec c$ *and* $b \not\prec a$*.*

In other words, a is e-relevant for δ if a asserts a condition that is protected in δ, or a conflicts with δ in the plan. a is e-relevant for Δ iff $\exists \delta \in \Delta$ such that a is e-relevant for δ. For example, the actions a, c and b in Fig.4.2 are e-relevant for Δ.

Definition 4.4 *a is* p-relevant *for δ iff $\delta = \langle p, b, a \rangle$ for some p and b.*

In other words, a is p-relevant for δ if a precondition of a is protected in δ. Likewise, a is p-relevant for Δ iff $\exists \delta \in \Delta$ such that a is p-relevant for δ. For example, the actions a, b and d are all p-relevant for Δ in Fig. 4.2.

Let P be a plan and Δ a set of protection intervals in P. The actions in P can be partitioned into four sets, $S(\Delta, P)$, $T(\Delta, P)$, $B(\Delta, P)$, and $I(\Delta, P)$, according to their relationship with Δ. These sets are defined as follows:

1. $S(\Delta, P) = \{a \mid a \in A(P)$, a is e-relevant for Δ, but not p-relevant for $\Delta\}$.

   Intuitively, $S(\Delta, P)$ consists of actions in P which only provide or delete the conditions for some protection intervals in Δ, but do not appear as a receiver of any protection intervals in Δ.

2. $T(\Delta, P) = \{a \mid a \in A(P)$, a is p-relevant for Δ, but not e-relevant for $\Delta\}$.

   Intuitively, $T(\Delta, P)$ consists of actions in P which only appear as receivers of some intervals in set Δ.

3. $B(\Delta, P) = \{a \mid a \in A(P)$, a is both p-relevant and e-relevant for $\Delta\}$.

   Intuitively, $B(\Delta, P)$ consists of actions in P which either appear both as providers and receivers of some intervals in Δ, or appear as receivers in Δ and deny some other conditions protected in Δ.

4. $I(\Delta, P) = \{a \mid a \in A(P)$, a is neither p-relevant nor e-relevant for $\Delta\}$.

   In other words, set $I(\Delta, P)$ consists of actions which are irrelevant to Δ.

As an example, consider the plan in Fig. 4.2. In that plan, $S(\Delta, P) = \{\ c\ \}$, $T(\Delta, P) = \{\ d\ \}$, $B(\Delta, P) = \{\ a,\ b\}$ and $I(\Delta, P) = \emptyset$.

Definition 4.5 *Let $\delta = \langle p, a, b\rangle$ be a protection interval for some a and b.* condition$(\delta) = p$.

Definition 4.6 *Suppose a is an action e-relevant for δ and $p =$* condition(δ).

$$\text{effects}_\delta(a) = \begin{cases} \{p\} & if\ p \in \text{effects}(a) \\ \{\neg p\} & otherwise \end{cases}$$

Definition 4.7 effects$_\Delta(a) = \bigcup_{\delta\in\Delta'}$ effects$_\delta(a)$, *where $\Delta' = \{\delta \in \Delta \mid a$ is e-relevant for $\delta\}$.*

Intuitively, $\text{effects}_\Delta(a)$ is the set of effects of a which are either protected in Δ, or conflict with some conditions protected in Δ. A similar definition can be given for the preconditions of a:

Definition 4.8 $\text{preconditions}_\Delta(a) = \bigcup_{\delta \in \Delta''}\{\text{condition}(\delta)\}$, *where* $\Delta'' = \{\delta \in \Delta \mid a$ *is p-relevant for* $\delta\}$.

The following syntactic restrictions are imposed on the sets of actions classified above. Let Φ be the set of reduction schemata for a planning system.

Restriction 4.1 *Given* $a \in S(\Delta, P)$, $\forall R \in \Phi$ *such that* R *is applicable to* a, $\exists a_m \in A(R(a))$ *such that*

1. $\text{effects}_\Delta(a) \subseteq \text{effects}(a_m)$, *and*
2. $\forall a_i \in A(R(a))$, *if* $a_i \neq a_m$, *then* $\text{effects}(a_i) \cap \text{effects}_\Delta(a) = \emptyset$.

Intuitively, Restriction 4.1 requires that an action in set $S(\Delta, P)$ has a unique "main" subaction, which asserts all of the effects of a relevant for Δ.

Lemma 4.1 *Let* $a \in S(\Delta, P)$ *be an action satisfying Restriction 4.1. Let* R *be a reduction schema applicable to* a. *Then if* P *is* Δ*-unlinearizable, then* $R(P)$ *is* $R(\Delta)$*-unlinearizable.*

Proof (by contradiction): Suppose $R(P)$ is $R(\Delta)$-linearizable. Let L' be a linearization of $R(P)$ which is $R(\Delta)$-consistent. From Restriction 4.1, there is a unique subaction a_m of a which asserts all the effects of a relevant for Δ, and none of the other actions assert any of such effects. Therefore, the only difference between Δ and $R(\Delta)$ is that a_m replaces a in every protection interval of Δ in which a appears. Let L'' be L' with all occurrences of $a_i \neq a_m$ deleted, $\forall a_i \in A(R(a))$. Since no subaction $a_i \neq a_m$ appears in $R(\Delta)$, L'' is $R(\Delta)$-consistent.

Now replace every occurrence of a_m in both L'' and $R(\Delta)$ by a, $R(\Delta)$ will be transformed to Δ, and the resultant sequence L must be Δ-consistent. This is because all of a_m's effects which are e-relevant to Δ are kept after this replacement, and any effects in $\text{effects}(a) - \text{effects}(a_m)$ are irrelevant and not harmful to Δ. However, L is a linearization of P. Therefore, P must be Δ-linearizable, contradicting to the assumption of the lemma. □

Restriction 4.2 *Given $a \in T(\Delta, P)$, $\forall R \in \Phi$ such that R is applicable to a, $\exists a_p \in A(R(a))$ such that*

1. $\text{preconditions}_\Delta(a) \subseteq \text{preconditions}(a_p)$, *and*

2. $\forall a_i \in A(R(a))$ *such that* $a_p \not\prec a_i$, $\text{effects}(a_i) \cap \text{effects}_\Delta(a) = \emptyset$.

Intuitively, this restriction requires that an action $a \in T(\Delta, P)$ has at least one subaction the preconditions of which include all the preconditions of a p-relevant for Δ.

Lemma 4.2 *Let $a \in T(\Delta, P)$ be an action satisfying Restriction 4.2. Let R be a reduction schema applicable to a. Then if P is Δ-unlinearizable, then $R(P)$ is $R(\Delta)$-unlinearizable.*

> **Proof** (by contradiction): Suppose that $R(P)$ is $R(\Delta)$-linearizable. Let L' be a linearization of $R(P)$ which is $R(\Delta)$-consistent. From Restriction 4.2, there is at least one subaction a_p of a which requires all the preconditions of a relevant for Δ, and none of the other actions assert any of such preconditions. Let a_p be any subaction of a which needs all of a's preconditions protected in Δ. In $R(P)$, every occurrence of a in Δ is replaced by a_p, and none of the $a_i \in A(R(a))$ appears in Δ if $a_i \neq a_p$. Therefore, deleting every subaction $a_i \neq a_p$ from L', the resultant sequence L'' is still $R(\Delta)$-consistent.
>
> Now replace every occurrence of a_p in $R(\Delta)$ by a. As a result, $R(\Delta)$ will be transformed to Δ. Also, remove all but one a_p from L'', and replace this a_p with a. The resultant sequence L must be Δ-consistent, since $\text{preconditions}_\Delta(a) \subseteq \text{preconditions}(a_p)$, and a does not deny any protected conditions in Δ. However, L is a linearization of P. Therefore, P must be Δ-linearizable, contradicting to the assumption of the lemma. □

Restriction 4.3 *Given $a \in B(\Delta, P)$, $\forall R \in \Phi$ such that R is applicable to Φ, $\exists a_m \in A(R(a))$ such that*

1. $\text{effects}_\Delta(a) \subseteq effects(a_m)$, *and* $\forall a_i \in A(R(a))$, *if* $a_i \neq a_m$, *then* $\text{effects}(a_i) \cap \text{effects}_\Delta(a) = \emptyset$, *and*

2. $\text{preconditions}_\Delta(a) \subseteq \text{preconditions}(a_m)$, *and* $\forall a_i \in A(R(a))$ *such that* $a_i \neq a_m$ *and* $a_m \not\prec a_i$, $\text{effects}(a_i) \cap \text{preconditions}_\Delta(a) = \emptyset$.

Intuitively, Restriction 4.3 requires that an action a in the set $B(\Delta, P)$ satisfies both Restrictions 4.1 and 4.2, and the "main" subactions in the two restrictions are identical.

Lemma 4.3 *Let $a \in B(\Delta, P)$ be an action satisfying Restriction 4.3. Let R be a reduction schema applicable to a. Then if P is Δ-unlinearizable, then $R(P)$ is $R(\Delta)$-unlinearizable.*

Proof (by contradiction): Suppose $R(P)$ is $R(\Delta)$-linearizable. Let L' be a linearization of $R(P)$ which is $R(\Delta)$-consistent. From Restriction 4.3, there is a unique subaction a_m of a which asserts all the effects of a relevant for Δ, which preconditions includes all the preconditions of a relevant for Δ, Let L'' be L' with all occurrences of $a_i \neq a_m$ deleted, where $a_i \in A(R(a))$. Similarly, let Δ'' be $R(\Delta)$ with all the intervals containing $a_i \neq a_m$ removed, where $a_i \in A(R(a))$. Then L'' is Δ''-consistent.

Now replace every occurrence of a_m in both L'' and Δ'' by a. Δ'' will be transformed to Δ, and the resultant sequence L must be Δ-consistent. This is because all of a_m's effects relevant to Δ are kept after this replacement, and any effects in effects(a) – effects(a_m) are irrelevant and not harmful to Δ. However, L is a linearization of P. Therefore, P must be Δ-linearizable, contradicting to the assumption of the lemma. □

The following theorem combines the results of the above three lemmas:

Theorem 4.1 *Let P be a plan and Φ be a set of action reduction schemata. Suppose the following conditions are satisfied:*

1. *every action in $S(\Delta, P)$ satisfies Restriction 4.1,*
2. *every action in $T(\Delta, P)$ satisfies Restriction 4.2, and*
3. *every action in $B(\Delta, P)$ satisfies Restriction 4.3.*

Let $Q(P)$ be any composite reduction of P produced by reducing the non-primitive actions in $A(P)$. Then if P is Δ-unlinearizable, then $Q(P)$ is $Q(\Delta)$-unlinearizable.

Proof: Let N be the number of non-primitive actions in P. Also let $Q(P)$ be any composite reduction of P produced by reducing i actions in $A(P)$,

where $i = 0, 1, \ldots, N$. We prove by induction that for any i, $Q(P)$ is $Q(\Delta)$-unlinearizable, where $Q(\Delta)$ is the corresponding reduction of Δ.

The base case corresponds to P before any reduction is done. Since P is Δ-unlinearizable, the theorem holds for this case. For the inductive hypothesis, assume that $Q(P)$ is a composite reduction of P produced by reducing $i < N$ actions in P, and that $Q(P)$ is $Q(\Delta)$-unlinearizable. Let a be a non-primitive action in both P and $Q(P)$. We will prove that $Q'(P)$ is $Q'(\Delta)$-unlinearizable, where $Q'(P)$ is produced from $Q(P)$ by reducing a.

The action a must belong to any of the four partitions of the actions in $Q(P)$. These partitions are $S(\Delta, P)$, $T(\Delta, P)$, $B(\Delta, P)$, and $I(\Delta, P)$. From the assumption of the theorem and Lemmas 4.1, 4.2 and 4.3, if a belongs to the first three sets, then $Q'(P)$ is $Q'(\Delta)$-unlinearizable. If a belongs to $I(\Delta, P)$, then a's effects and preconditions are not e-relevant nor p-relevant for $Q(\Delta)$. Therefore, reducing a will not affect the linearizability of $Q'(P)$ with respect to $Q'(\Delta)$. Thus, $Q'(P)$ must be also $Q'(\Delta)$-unlinearizable. □

Notice that the conditions of the theorem are only sufficient, and there may be schemata which do not satisfy the restrictions but still have the property that, when a plan P is reduced, it remains unlinearizable if P is. Thus, it is important to show that our restrictions are close to "minimum". Below, we show this through an example. Consider the plan in Fig. 4.3 (a). If a asserts both $\neg p$ and $\neg q$, then this plan is Δ-unlinearizable, where

$$\Delta = \{\langle p, c, b\rangle, \langle q, b, d\rangle\}.$$

In this plan P, a belongs to the set $S(\Delta, P)$. Suppose a can be reduced into $a_1 \prec a_2$, such that $\neg q$ is asserted by a_1 and $\neg p$ is asserted by a_2. Note that a does not satisfy Restriction 4.1, since the effects of a are distributed among its subactions. The reduction of P as a result of reducing a is shown in Fig. 4.3 (b). Note that Δ is unchanged by this reduction. The plan in Fig. 4.3 (b) is now Δ-linearizable. In particular, the linearization $c \prec a_1 \prec b \prec a_2 \prec d$ is Δ-consistent.

If P contains unresolvable conflicts, and if the actions in P satisfy the restrictions imposed on them, then according to the theorem, the conflicts remain unresolvable even if the actions in P are reduced. From this theorem, it is possible to define the relationship

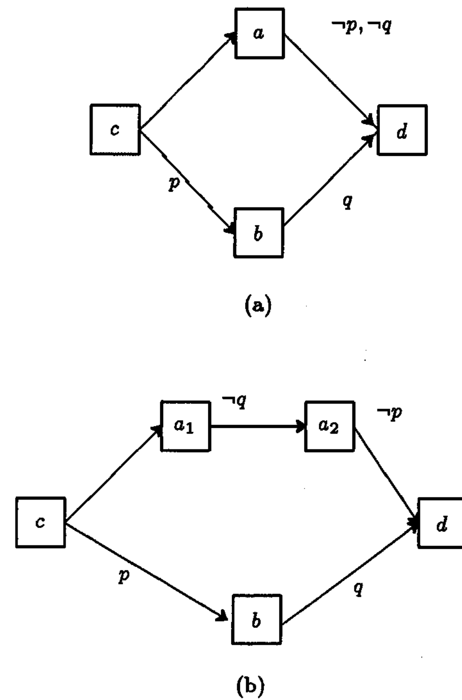


Figure 4.3: Resolving unresolvable conflicts in (a) by reduction to (b).

between a non-primitive action and its set of subactions precisely. These criteria enable one to check for the desired properties *before* planning starts. In the next section, we discuss how this can be done using the results developed in this section.

4.3 Downward-Unlinearizability

4.3.1 The "Unique Main Subaction" Restriction

In this section, a set of restrictions is provided which restricts the way an action relates to its set of subactions in its reduction. This set of restrictions enable one to check and possibly modify a given set of reduction schemata. Whenever the restrictions are satisfied, backtracking from a plan with a set of unresolvable conflicts will not lose any potential solutions at any level of reduction below.

Definition 4.9 *A set Φ of action reduction schemata is* downward-unlinearizable *iff the following condition holds. Let P be a plan and Δ be a set of protection intervals in P. Also let $Q(P)$ be any composite reduction of P using the action reduction schemata in Φ, and $Q(\Delta)$ be the corresponding reduction of Δ. Then if P is Δ-unlinearizable then $Q(P)$ is $Q(\Delta)$-unlinearizable.*

Our method for checking if a given set of reduction schemata is downward-unlinearizable is to develop a set of sufficient syntactic restrictions upon action templates in Λ and reduction schemata in Φ, based on Theorem 4.1.

Consider the following restriction:

Restriction 4.4 *Let a be an action and R be a reduction schema applicable to a. Reduction $R(a)$ satisfies Restriction 4.4 iff $\exists a_m \in A(R(a))$ such that*

1. $\text{effects}(a) \subseteq \text{effects}(a_m)$ *and* $\forall a_i \in A(R(a))$*, if* $a_i \neq a_m$*, then* $\text{effects}(a_i) \cap \text{effects}(a) = \emptyset$.

2. $\text{preconditions}(a) \subseteq \text{preconditions}(a_m)$*, and* $\forall a_i \in A(R(a))$ *such that* $a_i \neq a_m$*, if* $a_m \not\prec a_i$ *then* $\text{effects}(a_i) \cap \text{preconditions}(a) = \emptyset$.

This restriction will be referred to as the *Unique-Main-Subaction* Restriction. Intuitively, this restriction requires that the reduction $R(a)$ of an action a contains a "main" subaction a_m which asserts everything a asserts. In addition, the preconditions of a are required to "persist" till the beginning of a_m. A number of planning domains can be represented in ways that satisfy this restriction. For example, consider an action of fetching an object "object1" in a room "room1" (based on [48]). The action "fetch" can be reduced into a sequence of three subactions: getting into room1, getting near object1, and picking up object1, in that order. The third subaction, picking up object1, can be considered as the main subaction in this reduction. Notice that the condition that object1 is in room1 is a precondition of both the non-primitive action "fetch" and the main subaction. The complete reduction schema is given in table 4.1.

The following lemma can be easily verified:

Lemma 4.4 *Let Δ be a set of protection intervals associated with a plan P. Suppose $a \in A(P)$ is non-primitive, and $\forall R \in \alpha(a)$, $R(a)$ satisfies the Unique-Main-Subaction Restriction. Then*

action	preconditions	effects
fetch	Inroom(object1,room1)	Holding(robot1,object1)
achieve(Inroom(robot1,room1))	$\emptyset$	Inroom(robot1,room1)
achieve(Nextto(robot1,object1))	$\emptyset$	Nextto(robot1,object1)
pickup	Inroom(object1,room1)	Holding(robot1,object1)
	Inroom(robot1,room1)	
	Nextto(robot1,object1)	

Table 4.1: An action reduction schema satisfying the Unique-Main-Subaction Restriction.

1. *if* $a \in S(\Delta, P)$ *then* a *satisfies Restriction 4.1,*

2. *if* $a \in T(\Delta, P)$ *then* a *satisfies Restriction 4.2,*

3. *if* $a \in B(\Delta, P)$ *then* a *satisfies Restriction 4.3.*

Let Λ be the set of actions of a planning system, and Φ be the set of action reduction schemata.

Restriction 4.5 Φ *satisfies Restriction 4.5 iff for every reduction schema* $R \in \Phi$ *and every action template* $\mathbf{a} \in \Lambda$ *to which* R *is applicable,* $R(\mathbf{a})$ *satisfies the Unique-Main-Subaction Restriction.*

Theorem 4.2 *Every set of action reduction schemata satisfying Restriction 4.5 is downward-unlinearizable.*

> **Proof:** Let Φ be a set of action reduction schemata, and let P be a plan which is Δ-unlinearizable. We prove the theorem by induction on the number of reductions applied to P. Specifically, we prove that for any $i \geq 0$, $Q(P)$ is $Q(\Delta)$-unlinearizable, where $Q(P)$ is any composite reduction of P produced by reducing i non-primitive actions, using the reduction schemata in Φ.
>
> The base case, $i = 0$, corresponds to the plan P without any reduction. Since P is assumed Δ-unlinearizable, the theorem holds for the base case.
>
> For the inductive hypothesis, assume that $Q(P)$ is $Q(\Delta)$-unlinearizable, where $Q(P)$ is a composite reduction of P produced by reducing $n \geq 0$ actions.
>
> Let $Q'(P)$ be produced by reducing a non-primitive action in $Q(P)$, and let the corresponding reduction of $Q(\Delta)$ be $Q'(\Delta)$. Since every action in $Q(P)$ is an instance of some action in Λ and since Φ satisfies Restriction 4.5, from Lemma 4.4 the assumptions of Theorem 4.1 hold. Therefore, any reduction $Q'(P)$ of $Q(P)$ must be $Q'(\Delta)$-unlinearizable. Thus, the theorem holds for any composite reduction of P. □

Before discussing how this theorem can be used for checking the downward-unlinearizability property, we would like to comment on the applicability of the Unique-Main-Subaction

Restriction to planning domains. Intuitively, the Unique-Main-Subaction Restriction requires that every non-primitive action a has a unique main subaction a_m that asserts every effect of a, and requires every precondition of a as its precondition. Domains that satisfy this restriction have the following characteristics:

1. The goals to be achieved can always be broken down to several less complicated subgoals to solve. For each subgoal, a number of primitive actions are available for achieving it. Each such action requires several preparation steps before it can be performed, and a number of clean-up steps after it is done. This action can be considered as the main step in a reduction schema.

2. For each group of actions mentioned above, a hierarchy is built by associating with a non-primitive action a set of effects which are the purposes of the main step mentioned above, and a set of preconditions which are certain important preconditions of the main step.

A number of domains can be formulated in ways that satisfy the property of downward-unlinearizability. In Appendix A, we provide a representation of the planning knowledge of the blocks-world domain. In this domain, a number of blocks exist on the top of a table. A robot can move a block on the top of another, or onto the table. Each block can have at most one block on top of it, and a block can be on at most one other block. Condition On(x, y) represents that the block x is on top of another block y, Ontable(x) means that the block x is on the table, and Cleartop(x) means that block x has no other blocks on top of x. The top level actions are also the subgoals that the system knows how to solve. They are achieve(On(x, y)), achieve(Ontable(x)) and achieve(Cleartop(x)). More complicated goals can be composed from the conjunction of these.

It is not hard to verify that every reduction satisfies the Unique-Main-Subaction Restriction. For example, consider the reduction R_1 for achieve(On(x, y)). Notice that the in R_1(achieve(On(x, y))), the action makeon-block-1(x, y) is the main subaction, which asserts the effect On(x, y) of achieve(On(x, y)). In addition, the set of preconditions of achieve(On(x, y)) is empty, and neither achieve(Cleartop(x)) nor achieve(Cleartop(y)) asserts On(x, y). Thus, the Unique-Main-Subaction Restriction is satisfied. Likewise, the reduction R_5 for makeon-block-1(x, y) also satisfies this restriction, since the reduction contains only one subaction. For the rest of the reductions, the Unique-Main-Subaction

Restriction can be similarly verified.

As another example, consider a domain in which there is a room, a ladder, supplies for painting as well as a robot whose goal is to paint portions of the room and/or the ladder. Part of the representation for this domain has been given in Chapter 3. Suppose in addition to the actions and reduction schemata for painting, the robot also knows how to fetch an object using the action 'fetch' in Table 4.1). Represented in this way, every action reduction schema satisfies the Unique-Main-Subaction Restriction. For instance, the 'apply-paint-to-ceiling' step in the reduction of 'Paint(Ceiling)' is the main subaction. Also, the 'pickup' step in the reduction of 'fetch' is also a main subaction. Moreover, these reduction schemata all satisfy the Unique-Main-Subaction Restriction.

Restriction 4.5 does not depend on any particular plan, and it enables one to verify its validity for any given set of action templates Λ and action reduction schemata Φ. This verification process can be done before planning starts. The algorithm for verification is given in the next section.

4.3.2 Checking for Downward-Unlinearizability

The algorithm for preprocessing a set of action reduction schemata makes use of Restriction 4.5. This algorithm is run after the planning knowledge for a domain is defined, taken as input the set of action reduction schemata Φ and action templates Λ. Two goals will be achieved through preprocessing: (1) to see if the set of action reduction schemata is downward-unresolvable, and (2) if not, try to modify the schemata so that they have this property. Let Φ be a set of action reduction schemata and Λ be a set of actions. Together Φ and Λ provides the planning knowledge for a system. The algorithm for preprocessing is presented below. It returns True if Φ satisfies the restrictions imposed in the previous sections. If so, the set Φ of schemata satisfies the downward-unlinearizability property.

Algorithm *Checking.*

begin

$B := \Lambda$;

while $B \neq \emptyset$ **do**

$\mathbf{a} := \text{pop}(B)$;

if $\forall R \in \alpha(\mathbf{a})$, $R(\mathbf{a})$ satisfies the Unique-Main-Subaction Restriction **then**

Mark **a**;

$B := B - \{\mathbf{a}\}$;

end; (while)

if all the action templates in Λ are marked **then**

return(True)

else return(False)

end

To check if $R(\mathbf{a})$ satisfies the Unique-Main-Subaction Restriction, first check to see if there is an unique main subaction in $A(R(\mathbf{a}))$ that asserts every effect **a** asserts and has in its set of preconditions every precondition of **a**. If such a main subaction a_m exists, then check for every other subaction a_i in $A(R(\mathbf{a}))$

1. $\forall p \in$ preconditions($\mathbf{a}$), $\langle p, a_i, a_m \rangle \notin C(R(\mathbf{a}))$, and
2. effects(a_i) $\cap$ effects((a) $= \emptyset$.

To check each $R(\mathbf{a})$ for Unique-Main-Subaction Restriction, the above algorithm suggests a worst case time complexity of $O(|\Lambda|)$, where $|\Lambda|$ is the number of actions in Λ. Let k be the maximum number of action reduction schemata applicable to an action template. Then Algorithm *Checking* has a worst case complexity of $O(k \times |\Lambda|^2)$.

If the algorithm *Checking* returns True, then the set Φ of reduction schemata satisfies the downward-unresolvability restriction. In which case, whenever a hierarchical planner detects unresolvable conflicts in a plan, it does not have to consider the reduction of the plan as a means of resolving the conflicts.

4.3.3 Modifying Action Reduction Schemata

If algorithm *Checking* returns False for Φ, then in some cases it might be possible to modify the schemata, so that Φ is made to satisfy the restrictions of previous sections.

For example, let $\mathbf{a}$ be an action template with a precondition p and an effect q. Suppose $R(\mathbf{a})$ consists of two subactions a_1 followed by a_2, such that the precondition of a_1 is p, and the effect of a_2 is q (see Fig. 4.4 (a)). If $\langle \neg p, a_1, a_2 \rangle$ is a protection interval associated with $R(\mathbf{a})$, then clearly the Unique-Main-Subaction Restriction is violated. One way to modify this reduction schema is to remove the precondition p from effects($\mathbf{a}$), so that a_2 becomes the main subaction of a. As a result, the modified reduction R' satisfies the Unique-Main-Subaction Restriction. This new reduction schema is shown in Fig. 4.4 (b).

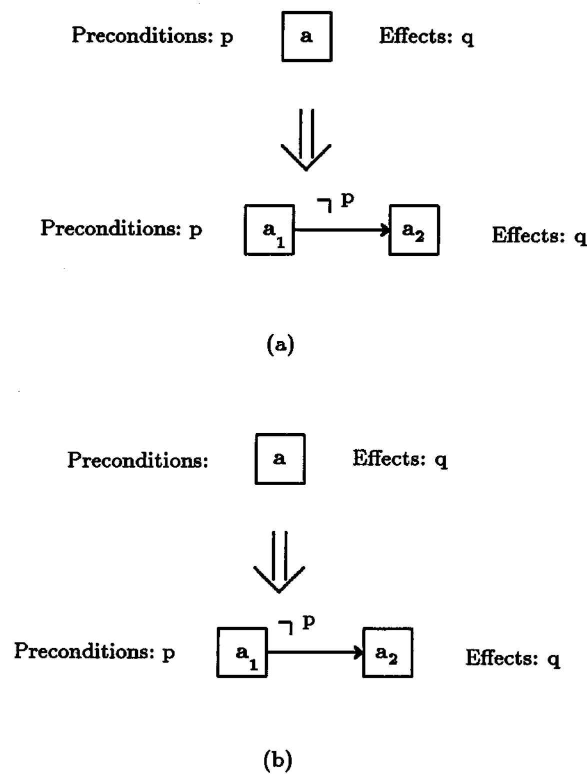


Figure 4.4: Modifying a reduction schema (a) by removing some preconditions of an action template (b).

The above example shows how modify a reduction schema by removing the preconditions of an action template. As another example, let $\mathbf{a}$ be an action template with two effects q and g such that the subaction a_1 of a has a effect q while another subaction a_2 has the effect g (Fig. 4.5 (a)). In this case, moving either g or q from the effects set of $\mathbf{a}$ will make the reduction schema satisfy the Unique-Main-Subaction Restriction . This is

shown in Fig. 4.5 (b).

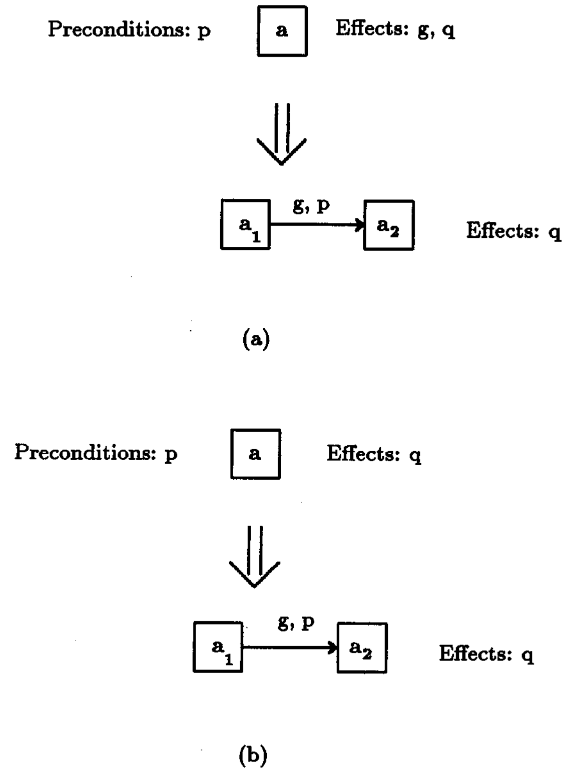


Figure 4.5: Modifying a reduction schema (a) by removing effects of action template **a** (b)

Unfortunately, the above examples do not suggest any general procedures for modifying a given set of action reduction schemata. This is because changing the representation of the action templates in Λ may make them less expressive than before. This can occur when, for example, some effects of an action are removed from its effects set. In extreme cases, changing the representation of an action template may cause planning to be less efficient than before. For example, removing the preconditions from an action may delay the detection of deleted-condition conflicts with the removed conditions, and as a result, more reasoning may be needed to fix those interactions after the modification. On the other hand, there may exist classes of planning problems for which a particular kind of schemata modification method exists, and finding out the characteristics of these domains is one of our future research directions.

4.4 Preprocessing Individual Action Template

Restriction 4.5 requires that for every action template in Λ, all of its reductions satisfy the Unique-Main-Subaction Restriction. Sometimes not all the action templates in Λ fit this requirement, but only some of them do. In situations like this, it may still be possible to ensure that unresolvable conflicts in a plan cannot be resolved in any of the plan's reduction.

For example, if P is a plan which is Δ-unlinearizable. Suppose for all the actions a in P, all the reductions of their corresponding action templates in Λ and their "descendents" satisfy the Unique-Main-Subaction restriction, then from Theorem 4.1, any composite reduction $Q(P)$ of P is $Q(\Delta)$-unlinearizable.

Restriction 4.6 *Let Λ be the set of action templates and Φ be the set of action reduction schemata of a planning system's planning knowledge. Let $\mathbf{a} \in \Lambda$ be an action template. $\forall \mathbf{a}' \in \mathrm{TC}(\mathbf{a})$ and $\forall R \in \alpha(\mathbf{a}')$, $R(\mathbf{a}')$ satisfies the Unique-Main-Subaction Restriction.*

The following theorem can be easily proved as a corollary to Theorem 4.1.

Theorem 4.3 *Let P be a plan Δ-unlinearizable. If for every action a in the plan,* template(a) *satisfies Restriction 4.6, then $Q(P)$ is $Q(\Delta)$-unlinearizable for any composite reduction Q.*

The set of action templates in Λ can be preprocessed in a way similar to the previous section. First, Algorithm *Checking* can be performed on Λ. If it returns False, then the following processing can be done: for each action template $\mathbf{a} \in \Lambda$, if every action template in $\mathrm{TC}(\mathbf{a})$ is marked by Algorithm *Checking*, then $\mathbf{a}$ satisfies Restriction 4.6. Mark all the action templates in Λ which satisfies Restriction 4.6. Suppose P is a plan unresolvable with respect to its set of protection intervals. If for every non-primitive action a in P, template(a) is marked in the above sense, then from Theorem 4.1, no composite reduction of P is linearizable with respect to its set of protection intervals. Therefore, backtracking from P will not lose any possible solutions.

4.5 Augmented Reduction of Protection Intervals

The downward-unlinearizability property requires that $Q(P)$ is $Q(\Delta)$-unlinearizable whenever P is Δ-unlinearizable. The protection intervals in $Q(P)$ consists of $Q(\Delta)$, as well as a set of new protection intervals associated with the reductions. In some cases, it is possible that although $Q(P)$ is $Q(\Delta)$-linearizable, $Q(P)$ is not linearizable with respect to the union of $Q(\Delta)$ and the set of new protection intervals.

Let P be a plan and

$$Q(P) = R_1(R_2(\ldots(R_n(P))\ldots)$$

be a composite reduction of P. Also let Δ be a set of protection intervals associated with P, and $Q(\Delta)$ be the corresponding composite reduction of Δ. Let C be the set of protection intervals associated with the reduction schemata R_i. Then the *augmented composite reduction* of Δ is

$$\mathrm{AU}_Q(\Delta) = C \cup Q(\Delta).$$

In this section, we look for restrictions that guarantee the following property: if P is Δ-unlinearizable, then $Q(P)$ is $\mathrm{AU}_Q(\Delta)$-unlinearizable.

Consider a blocks-world example where there are three parallel actions put-block-on-block(A, B), put-block-on-block(B, C) and put-block-on-block(C, A) (see Fig.4.6 (a)). Assume that three robot hands are available, which are designated as H_1, H_2 and H_3. Let i be the initial situation, and

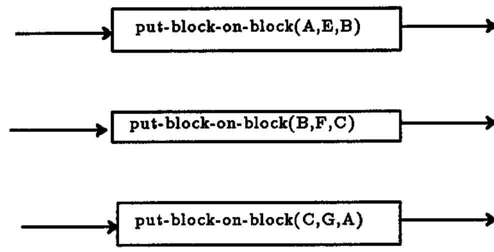
$$\begin{aligned}\Delta = \{ \quad & \langle \text{Cleartop}(A), i, \text{put-block-on-block}(A, B)\rangle, \\ & \langle \text{Cleartop}(B), i, \text{put-block-on-block}(B, C)\rangle, \\ & \langle \text{Cleartop}(C), i, \text{put-block-on-block}(C, A)\rangle\}.\end{aligned}$$

Let the plan in Fig.4.6 (a) be P. Then P is Δ-unlinearizable.

Now consider the composite reduction $Q(P)$ of P in Fig.4.6 (b), with

$$\begin{aligned}Q(\Delta) = \{ \quad & \langle \text{Cleartop}(A), i, \text{grab}(A)\rangle, \\ & \langle \text{Cleartop}(B), i, \text{grab}(B)\rangle, \\ & \langle \text{Cleartop}(C), i, \text{grab}(C)\rangle\}.\end{aligned}$$

Since it is now possible to interleave the subactions of the actions in P, $Q(P)$ is $Q(\Delta)$-linearizable by imposing ordering constraints as in Fig.4.7. However, $Q(P)$ is not lineariz-



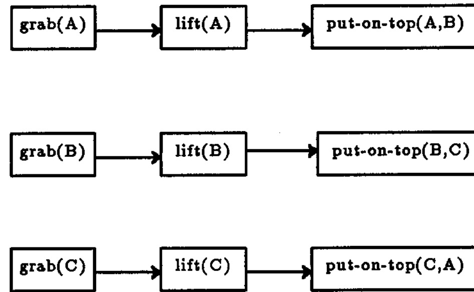


Figure 4.6: A plan with unresolvable conflicts.

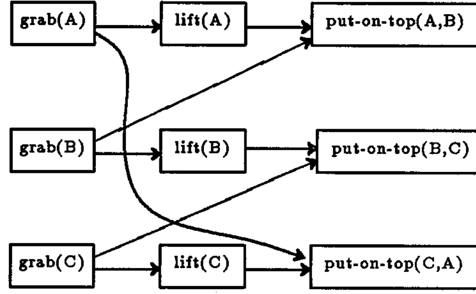


Figure 4.7: A plan is $\mathrm{AU}_Q(\Delta)$-unlinearizable.

able with respect to the set of all protection intervals associated with it. This is because $Q(P)$ is also associated with

$$\begin{aligned}\{\quad &\langle \text{Holding}(H_1, A), \text{grab}(A), \text{put-on-top}(A, B)\rangle,\\ &\langle \text{Holding}(H_2, B), \text{grab}(B), \text{put-on-top}(B, C)\rangle,\\ &\langle \text{Holding}(H_3, C), \text{grab}(C), \text{put-on-top}(C, A)\rangle\}.\end{aligned}$$

Let $\mathrm{AU}_Q(\Delta)$ be the union of $Q(\Delta)$ and the above set of protection intervals. Since a robot cannot hold a block when another block is on top of it, it is not hard to see that $Q(P)$ is $\mathrm{AU}_Q(\Delta)$-unlinearizable.

Consider the following restriction:

Restriction 4.7 *A reduction $R(a)$ satisfies Restriction 4.7 iff $\exists a_m \in A(R(a))$ such that*

1. $\text{effects}(a_m) \subseteq \text{effects}(a)$,

2. $\forall a_i \in A(R(a))$, *if a_i denies conditions in* $\text{effects}(a)$ *then* $a_i \prec a_m$,

3. $\forall p \in \text{preconditions}(a)$, *either of the following is true:*

   (a) $p \in \text{preconditions}(a_m)$ *and* $\forall a_j \in A(R(a))$, $\langle p, a_j, a_m\rangle$ *is not a protection interval in $C(R(a))$, or*

   (b) $\exists a_i \in A(R(a))$ *such that*

      i. $a_i \prec a_m$ *in $R(a)$,* $p \in \text{preconditions}(a_i)$,

ii. $\exists p' \in$ preconditions(a_m) *such that* $\langle p', a_i, a_m \rangle$ *is a protection interval in* $C(R(a))$*, and*

iii. p' *satisfies the following condition:* $\forall b \in \Lambda$*, if* $b \neq a_i$*,*

$$(\text{effects}(b) \supset \neg p) \supset (\text{effects}(b) \supset \neg p')$$

where "$\supset$*" is logical implication.*

Restriction 4.7 is different from Unique-Main-Subaction Restriction in part 3.(b). In Restriction 4.7, not all the preconditions are needed by the main subaction of a. Notice that the action put-block-on-block(x, y) in the blocks-world example does not satisfy the Unique-Main-Subaction Restriction, but it satisfies Restriction 4.7. For example, a_m is put-on-top(x, y), a_i is grab(x), p is Clear(x), and p' is Holding(H_i, x). Restriction 4.7 is satisfied because

$$\forall x, y, z, h.(\text{effects}(puton(y, z)) \supset \neg\text{Clear}(x)) \supset (\text{effects}(puton(y, z)) \supset \neg\text{Holding}(h, x)).$$

Theorem 4.4 *Suppose* $R(a)$ *satisfies Restriction 4.7. Then if* P *is* Δ*-unlinearizable, then* $R(P)$ *is* $\text{AU}_R(\Delta)$*-unlinearizable.*

Proof (by contradiction): Let L be a linearization of $R(P)$ $\text{AU}_R(\Delta)$-consistent , where $R(P)$ is obtained by reducing an action a in P. Let G_R be the set of a_i satisfying 3.(b) in Restriction 4.7. Below, we show provide a transformation which converts L into a linearization of P and $\text{AU}_R(\Delta)$ to Δ, and show that this transformation does not violate any protection interval of Δ in L. The transformation is defined as follows:

1. Let Σ be $A(R(a)) - (G_R \cup \{a_m\})$. Eliminate all the actions in Σ from L. Also for any action a' in the set $G_R \cup \{a_m\}$, remove every the precondition of a' which is achieved by any actions in Σ. Let the new sequence be L'. Remove every interval in $\text{AU}_R(\Delta)$ which contains an action $a_i \in \Sigma$, obtaining $\text{AU}'_R(\Delta)$. After this step, the new sequence L' is $\text{AU}'_R(\Delta)$-consistent. This is because removing subactions from in Σ from both $AU_R(\Delta)$ and L only removes protection intervals, and no new conflicts between the actions are created.

2. For every action $b \in A(P)$ such that $a_i \prec b \prec a_m$ in L', where $a_i \in G_R$, relocate b to the position immediately before the first a_i in L'. Repeat this step for all such b in a left to right order. After all the relocations are done, merge all the subactions in $G_R \cup \{a_m\}$ into a. After this step, there is still no new conflicts introduced. This is because according to the second part of Restriction 4.7, if b denies a precondition p of $a_i \in G_R$, then it would have violated the protection interval $\langle p', a_i, a_m \rangle$, because (effects($b$) $\supset \neg p$) $\supset$ (effects(b) $\supset \neg p'$). Let the resultant sequence after this modification be L''.
3. Replace every occurrence of $a_i \in (R(R(a)) \cup \{a_m\}$ in $\mathrm{AU}'_Q(\Delta)$ by a.

After the above modification steps, L is transformed to a linear sequence L'' of the actions in P. Similarly, the set of protection intervals $\mathrm{AU}'_Q(\Delta)$ is transformed to Δ. Since no protection interval violations are created after the transformations, L'' must be Δ-linearizable, violating the assumption of the theorem. Therefore a contradiction is derived. □

The following corollary can be easily proved by induction using the previous theorem.

Corollary 4.1 *Let Φ be a set of action reduction schemata. Suppose $\forall \mathbf{a} \in \Lambda$, $\forall R \in \alpha(\mathbf{a})$, $R(\mathbf{a})$ satisfies Restriction 4.7. Then for for any plan P with protection intervals Δ, and any composite reduction $Q(P)$ of P produced by the reduction schemata in Φ, if P is Δ-unlinearizable then $Q(P)$ is $\mathrm{AU}_Q(\Delta)$-unlinearizable.*

Preprocessing can be done in a similar manner as Algorithm *Checking* for the whole set of action templates in Λ, or as done in the previous section for individual action templates. Because of step $(b).iii$ of Restriction 4.7 checking for each action reduction schemata takes $O(|\Lambda|^2)$. Thus, the total time complexity for checking all the action templates and reduction schemata is $O(k \times |\Lambda|^3)$, where k is the maximum number of action reduction schemata applicable to an action template.

4.6 Point-Protected Conditions

4.6.1 Definitions

In the previous sections, we have considered unresolvable conflicts among the actions in a plan, which contains a set of conditions that have to be protected throughout intervals of time. This type of condition can be referred to as *interval-protected conditions.* There is another kind of condition in planning which requires only the validity of a set of conditions right before an action is started. This type can be called *point-protected conditions.*

Definition 4.10 *A point-protected condition γ is a pair: $\langle p, a\rangle$, where p is a condition which has to hold immediately before the action a.*

Let P be a plan and $\gamma = \langle p, a\rangle$ be a point-protected condition in P. Suppose that the reduction schema R is applicable to a. Let $R(P)$ be the plan which results when a is replaced by $R(a)$ in P. Then $R(P)$ has one or more point-protected conditions derived from γ which involve the subactions $a_i \in A(R(a))$. In particular, let a_i be a subaction of a in $A(R(a))$ such that $p \in \text{preconditions}(a_i)$. Then $R(P)$ will have a point-protected condition $\langle p, a_i\rangle$.

Let $Q(P)$ be a composite reduction of P. If Γ is a set of point-protected conditions associated with P, then the corresponding composite reduction of Γ is denoted by $Q(\Gamma)$.

Definition 4.11 *Let Γ be a set of point-protected conditions. A sequence L of actions is* Γ-consistent *if for every element $\langle p, a\rangle$ of Γ, $\exists b$ in L such that*

1. $b \prec a$ *in* L,

2. $p \in \text{effects}(a)$, *and*

3. $\forall c$ *such that* $a \prec c$ *and* $c \prec b$, $\neg p \notin \text{effects}(c)$.

Definition 4.12 *A plan P is* Γ-unlinearizable *iff no linearization of P is Γ-consistent.*

Suppose a plan P is Γ-unlinearizable. $R(P)$ may be $R(\Gamma)$-linearizable. This can occur when there is a subaction w of an action in the plan P which achieves the precondition of some of the actions in the conflict, and thus resolves it. Such actions has been referred to as white knights[7]. For instance, suppose there is a double cross conflict involving actions a and b. A white knight in this situation could be an action w such that

1. effects(w) $\supset$ preconditions(a),

2. effects(b) $\not\supset$ $\neg$preconditions(w) and

3. effects(w) $\not\supset$ $\neg$effects(b).

Therefore the conflict can be resolved by the ordering $b \prec w \prec a$. The action w can be a subaction of some existing action c in the plan. Because w only appears in some reduction of the current plan, an unresolvable conflict in P can be resolved in $Q(P)$. As in the previous sections, we will develop restrictions under which such cases do not occur.

4.6.2 Strong Unresolvability

The following definition characterizes a special class of conflicts involving pointed-protected conditions in Γ.

Definition 4.13 *P is said to be* Γ-strongly-unresolvable *if by definition P is Γ-unlinearizable, and $\forall \langle p, b\rangle \in \Gamma$, the following condition holds: $\forall \mathbf{a} \in \Lambda$ and $\forall q \in$ effects($\mathbf{a}$), $q\beta \neq p$ for all the substitution β.*

Intuitively, a plan is Γ-strongly-unresolvable if P is Γ-unlinearizable, and no action can assert any conditions protected in Γ.

As an example, consider the plan in Fig. 4.8. Suppose there are no actions in the action reduction schemata which assert either p or q. Let Γ be

$$\{\langle p, a\rangle, \langle q, b\rangle\}.$$

Then the plan is Γ-strongly-unresolvable.

Theorem 4.5 *Let P be a plan and Γ be a set of point-protected conditions in P. Suppose the set of reduction schemata Φ of a planning system is downward-unlinearizable. Then if P is Γ-strongly-unresolvable, then any composite reduction $Q(P)$ of P is $Q(\Gamma)$-unlinearizable.*

> **Proof:** Since Φ is downward-unlinearizable, the only situation in which $Q(P)$ is $Q(\Gamma)$-linearizable is when $Q(P)$ contains one or more actions w not in $A(P)$,

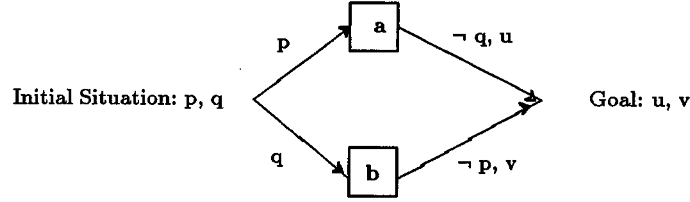


Figure 4.8: An example of strongly-unresolvable-conflicts.

such that w asserts some conditions protected in Γ. But this cannot happen, because P is Γ-strongly-unresolvable and therefore no action asserts any conditions in Γ. Thus, $Q(P)$ must remain Γ-unlinearizable. □

Notice that the condition of strong-unresolvability is more strict than unresolvability of conflicts in general, since it requires that no actions which can assert any conditions in Γ exist at the time when the conflicts occur. To see why this requirement is needed intuitively, consider the following example.

Consider a plan containing three actions a, b, and c unordered with each other (Fig. 4.9 (a)). Suppose their preconditions and effects are

$$\text{preconditions}(a) = \{p\}, \text{effects}(a) = \{\neg q\},$$
$$\text{preconditions}(b) = \{q\}, \text{effects}(b) = \{\neg p\},$$
$$\text{preconditions}(c) = \{e\}, \text{effects}(c) = \{\neg e\}.$$

Then the current plan is Γ-unlinearizable, where Γ is

$$\{\langle p, a\rangle, \langle q, b\rangle, \langle e, c\rangle\}.$$

Suppose a reduction of c contains two subactions in a sequence $v \prec w$, with

$$\text{preconditions}(w) = \{e\}, \text{effects}(w) = \{\neg e, q\}$$

(Fig. 4.9 (b)). Then the reduced plan is now linearizable with a linearization $v \prec a \prec w \prec b$, although inserting w into the plan Fig. 4.9 (a) will still result in a set of unresolvable conflicts, which are between a and b, and between c and the inserted w.

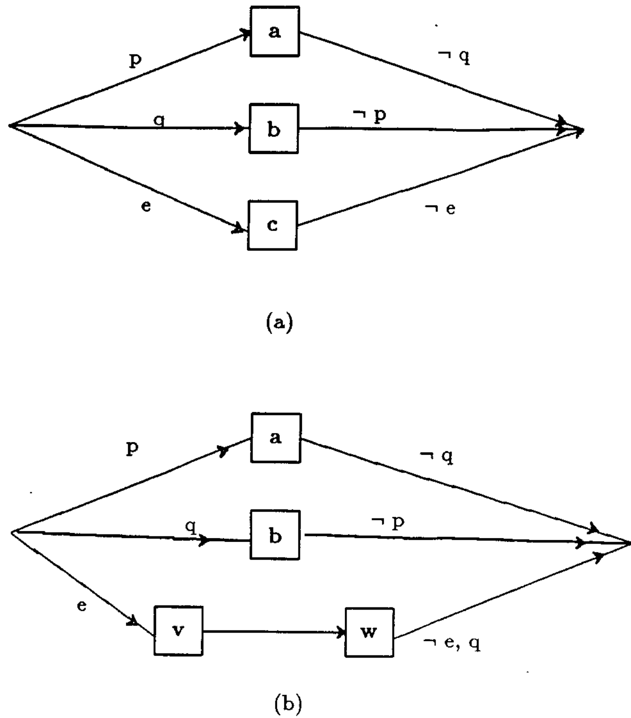


Figure 4.9: Resolving conflicts by reduction only.

In general, knowing whether P contains a set of conflicts that are unresolvable is very difficult. If an action a is inserted to restore some deleted conditions in the plan, the insertion of a may bring about new conflicts if a deletes the precondition of some other actions, or a's preconditions are deleted by some existing actions. In general, asking whether a plan is unresolvable is equivalent to asking whether a plan exists for achieving a given set of goals—an undecidable problem. This is part of the reason why we concentrate only on strong-unresolvable conflicts.

One way to efficiently tell if a plan is strongly unresolvable is to check all the conditions protected in Γ are *non-restorable*, i.e., all of these conditions are present only in an initial situation, none of which can be restored once it is deleted by some actions in the plan. One practical example is in the domain of automated manufacturing, where a machining part is produced by a sequence of metal cutting operations. If a piece of metal is cut off through one operation, in general it is impossible to put the piece back onto the stock. This type of interaction will be discussed in more detail in the second part of the thesis.

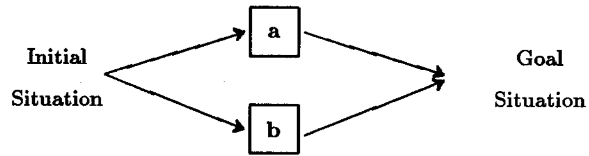
4.7 Other Kinds of Goal Interactions

In the previous sections, we have considered some formal properties of hierarchical planning systems, under the assumption that all goal interactions are deleted-condition conflicts. That is, the only way for the actions in a plan to interact with one another is to have one action's effects delete the other action's preconditions. When using these restrictions, one has to be careful when extending the results to domains where other types of interactions exist.

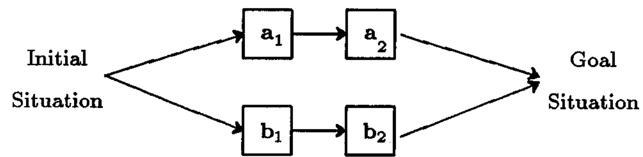
For example, consider a plan P consisting of two actions a and b (Fig. 4.10 (a)). Suppose in addition to the preconditions and effects, each action is also associated with a duration value, which is a lower bound on how long the action will take. Suppose the durations for both a and b are T. If there is a constraint that both a and b have to be completed in time $S < T$, then an unresolvable conflict occurs in P—no linearization can satisfy the constraint.

Suppose that a can be reduced to $a_1 \prec a_2$, and b can be reduced to $b_1 \prec b_2$ (Fig. 4.10 (b)). Suppose also that a_1 and b_1 can be replaced by another action c (Fig. 4.10 (c)), such that c can achieve the effects of both a_1 and b_1 within less time. If the new plan containing c, a_2 and b_2 requires less time than Φ, then the temporal conflict which is unresolvable in P is resolvable in $Q(P)$, although a and b may both satisfy the restrictions in the previous sections.

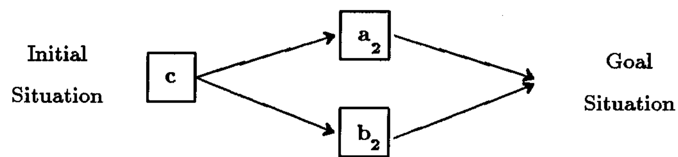
To see why this is the case, consider the types of interactions involved. The first new type of interaction can be referred to as a *temporal* interaction, in which there is a deadline as to when the plan has to be completed. The second type of interaction, the one which made it possible to resolve the conflicts at a level below, can be referred to as *action-merging interaction.* We will consider this type of interaction in much more detail in the later chapters.



(a)



(b)



(c)

Figure 4.10: Resolving a conflict by plan reduction and then merging.

Chapter 5

Ordering-Induced Independence

5.1 Motivation

One desirable effect of hierarchical planning is that important interactions between actions can be dealt with first, while unimportant ones are resolved later. Such interactions are often handled by introducing temporal orderings among the actions which interact. Ideally, handling the interactions in this way has the following effect: if a and b are actions at a higher level of reduction, and a time-ordering is assigned between them, then at the latter stages of planning, there is no need to check for conflicts between the subactions of a and the subactions of b. If this *ordering-induced independence* property can be satisfied by a and b, and if the ordering between them is not undone at a later stage, then an exponential amount of computational effort can be saved.

To illustrate this, consider a hierarchical planning problem with k levels of reduction, and suppose that each non-primitive action can be reduced into m subactions at the level below. Suppose the levels of reduction are numbered in ascending order from top-down, so that the bottom level is Level k and the top level is Level 0. A hierarchical nonlinear planner will finish removing all the goal interactions on the i'th level before it goes into

the $(i+1)^{st}$ level. An action at the i'th level will be reduced into m^i actions at Level k. Therefore, each time-ordering arc which can be introduced at the i'th level avoids $(m^{k-i})^2$ number of interaction checks at the k'th level. Thus, if the planner can add e time ordering arcs at each level of abstraction, the total amount of planning time saved will be

$$\sum_{i=0}^{k-1}(em^{2(k-i)}) = O(em^{2k}).$$

Knowing which actions satisfy the ordering-induced independence can provide a planner with a better control strategy. To take care of deleted-condition conflicts, a planner may have several alternate choices as to upon which actions certain precedence orderings should be imposed. A good decision is to choose the ones which satisfy the ordering-induced independence property.

As an example, consider part of a plan in Fig. 5.1. In this plan, there are two protection intervals $\langle p, a, b\rangle$ and $\langle \neg p, c, d\rangle$. They also create a conflict between the actions. To resolve the conflict, either b is ordered before c, or d is ordered before a. If the subactions of b will not delete the preconditions of c, and the subactions of c will not delete the useful effects of b, and also a and d do not have this property, then a sensible choice is to choose the ordering $b \prec c$.

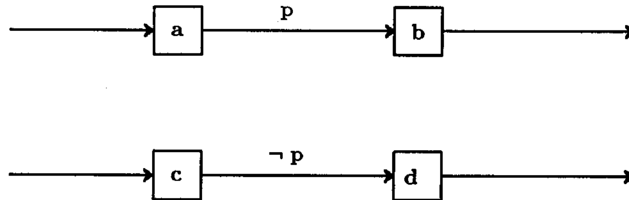


Figure 5.1: A plan in which ordering has to be imposed to resolve conflicts.

For a given problem domain, there may be many different ways to define a set of action reduction schemata. Some ways result in non-primitive actions which satisfy the ordering-induced independence property, while others may not. Below we consider syntactic restrictions that can ensure this property, and enable preprocessing for a given set of actions.

5.2 Imposing Syntactic Restrictions

If an ordering $a \prec b$ is assigned in the current plan, all the subactions of action a will precede that of action b. Therefore, at any later stages of plan reduction the effects of any of the subactions of b cannot delete the preconditions of the subactions of a, and the effects of the subactions of a cannot delete any of the effects of the subactions of b. However, it is possible that some subactions of a may delete some preconditions of b, and certain subactions of b may delete the effects of the action a which are protected.

Let a be a non-primitive action in a plan and $R(a)$ be a reduction of a. Since $R(a)$ is partially ordered, it may have more than one possible linearization. At the end of every linearization L, a set of conditions hold, which are asserted by the actions in L. We use effects($R(a)$) to denote the *union* of these sets of conditions, for all the linearizations. More formally,

$$\begin{aligned} \text{effects}(R(a)) \quad = \quad & \{p \mid \exists a_i \in A(R(a)) \text{ such that} \\ & p \in \text{effects}(a_i), \textit{ and} \\ & \forall b \in A(R(a)) \text{ such that } a_i \prec b, \neg p \notin \text{effects}(b)\}. \end{aligned}$$

Let $|R(a)|$ be the number of actions in $R(a)$, then effects($R(a)$) can be found in time $O(|R(a)|^3)$ by computing the transitive closure of the plan.

Definition 5.1 *Let P be a plan, and $a, b \in A(P)$ be actions in the plan. The* ordering-induced independence *property holds for (a, b) if the following conditions are satisfied: for any composite reductions Q_1 and Q_2 of a and b, respectively,*

1. $\forall p \in$ preconditions(b) *and* $\forall q \in$ effects($Q_1(a)$), *$\neg p$ and q are not unifiable.*

2. $\forall q \in$ effects(a) *such that $\langle q, a, c\rangle$ is a protection interval and $b \prec c$ in P, $\neg q$ is not unifiable with any $p \in$* effects($Q_2(b)$).

Notice that since p and q are literals, checking whether or not they are unifiable is straightforward. Thus, if (a, b) satisfies the ordering-induced independence property, and $a \prec b$ in a plan, then no subactions of a can conflict with the preconditions of b, and no subactions of b can conflict with the effects of a which has to persist over them.

Consider the following restrictions.

Restriction 5.1 *Let a and b be two actions. $\forall R \in \alpha(a)$ and $\forall p \in \text{effects}(R(a))$, if $\neg q \in \text{preconditions}(b)$ and p and q are unifiable, then $p \in \text{effects}(a)$.*

This restriction says that if any subaction a_i of a can possibly deny a precondition of b, then this effect of a_i should also be an effect of a. If this restriction is satisfied by actions a and b, then we say that a satisfies Restriction 5.1 with respect to b.

Restriction 5.2 *Let a and b be two actions. $\forall R \in \alpha(b)$ and $\forall q \in \text{effects}(R(b))$, if $\neg q \in \text{effects}(a)$ and p and q are unifiable, $q \in \text{effects}(b)$.*

This restriction says that if a subaction b_j of b can possibly deny an effect of a, then this effect of b_j should also be an effect of b. If a and b satisfies this restriction, then we say that a satisfies Restriction 5.2 with respect to b. If actions a and b satisfy both of these restrictions, then we say (a, b) satisfies the *Condition-Promotion Restriction.*

The following theorem allows one to check if a pair of actions satisfies the ordering-induced independence property.

Theorem 5.1 *The ordering-induced independence property holds for (a, b) if the following conditions hold.*

1. *every action template in* $\text{TC}(\text{template}(a))$ *satisfies Restriction 5.1 with respect to* $\text{template}(b)$, *and*
2. *every action template in* $\text{TC}(\text{template}(b))$ *satisfies Restriction 5.2 with respect to* $\text{template}(a)$.

We prove the first part of the conclusion of the theorem: Let $Q(a)$ be a composite reduction of a. Then $\forall p \in \text{preconditions}(b)$, $\neg p \notin \text{effects}(Q(a))$. The second part can be proved in a similar way.

Proof: The proof is done by induction on the level i of composite reduction $Q(a)$ of a, for $i \geq 0$. The base case $i = 0$ corresponds to the plan P. Since an order is assigned between a and b in P by a least-commitment planner, the effects of a cannot delete any precondition of b. Therefore, the base case holds.

For the inductive hypothesis, assume that $\forall p \in \text{preconditions}(b)$, $\neg p \notin \text{effects}(Q(a))$, where $Q(a)$ is any composite reduction of a produced by applying $n \geq 0$ reduction schemata to a.

Now consider the reduction $R(Q(a))$, produced by applying R to a non-primitive action a' in $Q(a)$. Suppose to the contrary of the theorem that $\exists p \in$ preconditions(b) such that $\neg p \in$ effects$(R(Q(a)))$. Let a_i be the descendent of a which asserts $\neg p$. Since $\neg p$ does not occur in effects$(Q(a))$, a_i can only be a subaction of a', that is, $a_i \in A(R(a'))$. Then $\neg p \in$ effects$(R(a'))$ for some $p \in$ preconditions(b) and $\neg p \notin$ effects(a'). This violates Restriction 5.1. Thus, a contradiction results. □

This theorem allows one to preprocess the planning knowledge of a planning system. If all the information is gathered before planning starts, then a planner can make decision more intelligently during planning, by using these information. Specifically, if the Condition-Promotion Restriction is satisfied by a and b, then the ordering $a \prec b$ will never be undone because of the interactions between the subactions of a and b.

5.3 Checking for "Ordering Induced Independence" Property

As in the case of downward-unlinearizability in the previous chapter, we would like to preprocess a given set of action reduction schemata, to check whether the actions in the schemata satisfy the Condition-Promotion Restriction. Again, the purpose of this process is (1) to recognize and mark pairs of actions that satisfy the condition promotion restriction, and (2) if a pair of actions does not satisfy the restriction, suggest ways to fix the definition of the reduction schemata so that they do.

Let $\mathbf{a}$ and $\mathbf{b}$ be two action templates in Λ. The following algorithm checks if $(\mathbf{a}, \mathbf{b})$ satisfy the Condition-Promotion Restriction:

Algorithm 5.3

1. Compute TC($\mathbf{a}$) and TC($\mathbf{b}$). This step can be done in time $O(k \times |\Lambda|^2)$ using depth first search, where k is the maximum number of reduction schemata applicable to an action.

2. Check for every action template $\mathbf{a_i} \in$ TC($\mathbf{a}$), $(\mathbf{a_i}, \mathbf{b})$ satisfies Restriction 5.1 with respect to $\mathbf{b}$. Notice that TC($\mathbf{a}$) can have at most $|\Lambda|$ elements. To check the restriction

for each pair $(\mathbf{a_i}, \mathbf{b})$, one can first compute $e = (\text{effects}(R(\mathbf{a_i})) - \text{effects}(a_i))$. $(\mathbf{a_i}, \mathbf{b})$ satisfies Restriction 5.1 if $\forall p \in e$ and $\forall q \in \text{preconditions}(\mathbf{b})$, p and $\neg q$ is not unifiable. If this is done for all the actions in TC($\mathbf{a}$), then the time complexity for this step is $O(E^2 \times |\Lambda|^2)$, where E is the maximum number of conditions in the preconditions or effects of an action.

3. Check if for every action template $\mathbf{b_j} \in \text{TC}(\mathbf{b})$, $(\mathbf{a}, \mathbf{b_j})$ satisfies Restriction 5.2. This step is similar to the previous one, and has a time complexity of $O(E^2 \times |\Lambda|^2)$.

The total time complexity for running this algorithm is $O((E^2+k) \times |\Lambda|^2)$=$O((E^2+|\Phi|) \times |\Lambda|^2)$.

If both the second and third steps of Algorithm 5.3 succeed, then the ordering-induced independence property holds for the pair $(\mathbf{a}, \mathbf{b})$. In that case, every pair actions which are instances of $\mathbf{a}$ and $\mathbf{b}$ also satisfy the property.

As an example, consider the action templates and their reduction schemata for the blocks-world domain, which are listed in Appendix A. Let $\mathbf{a}$ be **makeon-table-1**(x, y), and $\mathbf{b}$ be **makeon-block-1**(u). Then

$$\begin{aligned}
&\text{TC}(\mathbf{a}) = \{\textbf{put-block-on-table}(u, v)\},\\
&\text{TC}(\mathbf{b}) = \{\textbf{put-block-on-block}(x, z, y)\},\\
&\text{preconditions}(\mathbf{a}) = \{\text{Block}(u), \text{Cleartop}(u)\},\\
&\text{preconditions}(\mathbf{b}) = \{\text{Block}(x), \text{Block}(y), \text{Cleartop}(x), \text{Cleartop}(y)\},\\
&\text{effects}(\mathbf{a}) = \{\text{Ontable}(u)\},\\
&\text{effects}(\mathbf{b}) = \{\neg\text{Cleartop}(y), \text{On}(x, y)\}.
\end{aligned}$$

Let e_1 be $\text{effects}(R_7(\mathbf{a})) - \text{effects}(\mathbf{a})$. Then

$$e_1 = \{\text{Cleartop}(v), \neg\text{On}(u, v)\}.$$

Also, let e_2 be $\text{effects}(R_5(\mathbf{b})) - \text{effects}(\mathbf{b})$. Then

$$e_2 = \{\text{Cleartop}(z), \neg\text{On}(x, z)\}.$$

Thus, no conditions in e_2 deny any conditions in preconditions($\mathbf{b}$), and no conditions in e_1 deny any conditions in effects($\mathbf{a}$). Therefore, the pair $(\mathbf{a}, \mathbf{b})$ satisfies the ordering-induced independence property.

During planning, choices have to be frequently made for imposing orderings among the actions. If two actions a and b satisfy the ordering-induced independence property, while the other pairs do not, then the ordering ordering $a \prec b$ should be favored, since no backtracking can be incurred as a result of the conflicts between the subactions of a and b.

As in the previous chapter, there may not be a general procedure available for modifying a set of reduction intervals. However, in some special cases, modifications to the schemata definitions can be made by "promoting" certain effects of actions to their parent actions in order to guarantee the ordering-induced independence property.

5.3.1 A Special Case Of Schemata Definition

A special case of action reduction schemata definition exists which allows one to simplify the algorithm for checking if the ordering-induced independence property holds for all the actions in a set of schemata. This special case occurs when all the actions in the set of action reduction schemata have only positive literals as their preconditions.

Restriction 5.3 *A non-primitive action a satisfies the* negation-promotion *restriction if by definition* $\forall R \in \alpha(a)$ *and* $\forall \neg p \in \text{effects}(R(a))$, $\neg p \in \text{effects}(a)$.

The result is given in the following corollary to theorem 5.1.

Corollary 5.1 *Let Φ be the set of action reduction schemata and Λ be the set of action reduction schemata for a planning system. Suppose for every action template* **a** *in Λ, all the preconditions of* **a** *are positive literals. If all the actions in Λ satisfy the negation-promotion restriction, then every pair of actions in the domain satisfies the ordering-induced independence property.*

Proof: Since every action template **a** in Λ satisfies the Negation-Promotion Restriction, and only negated propositions can deny any precondition of an action, it follows that any pair of actions must also satisfy the Condition-Promotion Restriction. Therefore, from Theorem 5.1, every pair of actions in Λ satisfies the ordering-induced independence property. □

The advantage of this corollary is that it allows a set of action reduction schemata to be checked for ordering-induced independence property in a more efficient manner. In particular, the Negation-Promotion Restriction only restricts the way each action is related to its own set of subactions in its reduction schema. For each element p of effects($R(a)$), if p is a negative literal, then a check can be made to see if p is also in effects(a). If two actions a and b satisfy the Condition-Promotion Restriction, then both (a, b) and (b, a) have the ordering-induced independence property.

Part II

Multiple Goal Plan Optimization with Limited Interactions

Chapter 6

Planning For Multiple Goals—Definitions

6.1 Problem Statement

In Part I we described conditions under which planning efficiency can be improved by preprocessing a planner's planning knowledge. Below, another approach to improving planning efficiency is presented. As discussed in Chapter 2, domain-dependent planning and domain-independent planning can be considered as two extremes on a spectrum of planning methods. An intermediate method is to restrict the types of goal interactions that can occur, and design planning algorithms for domains where no other types of interactions occur. This method will be called the *limited-interaction* planning approach.

Limited-interaction planning will be presented in the context of multiple goal planning problems. A multiple goal planning problem can be defined as follows. Given g goals $G_1, G_2, \ldots, G_g$ to be achieved, find a plan to solve all of them. Many planning problems can be considered as special cases of the multiple goal planning problem. As an example

in the domain of automated manufacturing, a machined part usually consists of several manufacturing features, where each feature can be considered as a individual goal, and the whole part as the conjunction of the goals.

In general, the problem is difficult due to the interactions between the goals. An action for achieving one goal may interfere with other actions for achieving other goals. Some of the interactions between goals are harmful, while others are useful. Detecting and handling these interactions has been very inefficient in general. A planner planning for multiple goals has to check for interactions among the actions every time an action is inserted. Without knowing what kinds of interactions are allowed, and what kinds of restrictions the interactions obey, a planner has to spend a large amount of computation on handling the interactions.

However, in many practical domains the goals are only loosely coupled with each other. In other words, the interactions between the goals are limited. In the following, we will try to characterize what is meant by "limited interactions", and propose an efficient approach to handle multiple goal interactions when there are limited interactions between the goals. Specifically, our approach is to develop one or more plans for each goal individually, and then combine them together. During this combination phase the interactions between the goals are detected and handled. The reason for using this approach is to achieve more efficiency than conventional approaches, in which the goals are achieved all together.

To see this, let $G_1, G_2, \ldots, G_n$ be the goals to be achieved. For each goal G_i, let b_i be its branching factor and d_i its depth [1]. If the goals are solved conjunctively, the size of the search space is

$$O\left(\left(\sum_{i=1}^{n} b_i\right)^{\sum_{i=1}^{n} d_i}\right),$$

which is $O((nb)^{nd})$, where $b = \max_{i=1}^{n} b_i$, and $d = \max_{i=1}^{n} d_i$.

On the other hand, if we can solve the goals individually and then combine the solutions to form a global plan, the complexity of the method will be

$$O(nb^d + T),$$

where T is the complexity for identifying and handling interactions between the plans. Moreover, in cases where there are several alternative plans for each goal, T will also

[1] these notations are adopted after [20]

include an additional complexity for choosing one plan for each goal in order to satisfy certain optimality criteria. In general, T is still exponential. However, when the goal interactions are limited, we can construct branch-and-bound algorithms with good heuristics for combining and optimizing the plans. As a result, the total time complexity can be reduced. We demonstrate this claim in the following sections.

As mentioned earlier, one of the major challenges is to characterize "limitedness" of the interactions between the goals. The specific questions to ask are then (1) what kinds of interactions do we allow in our domains of application? (2) for those interactions which are hard to handle, what kinds of restrictions can we impose so that the resulting problem is well-behaved enough for us to handle? Notice that the restrictions should also allow the planning methods to be applicable to a large enough problem domains to be interesting.

Below, a plan is defined in the same way as in Part I. In addition, we allow actions to have costs, and the cost of a plan is the sum of the costs of the actions. A plan can also be associated with a set of constraints, such as one that requires certain actions to occur at the same time.

6.2 Types of Inter-Goal Interactions

Depending on what kinds of interactions occur among the actions in the plans, it might or might not be possible for all of the goals to be achieved. In this and the next two chapters, we consider only the following kinds of interactions.

1. An *action-precedence* interaction is an interaction which requires that an action a in some plan P_i must occur before an action b in some other plan P_j. This can occur, for example, if b removes one of the preconditions necessary for a, and there is no other action which can be inserted after b to restore this precondition.

   Much previous work in planning has dealt with deleted-condition interactions. Some action-precedence interactions are expressible as deleted-condition interactions, and conversely, some deleted-condition interactions can be resolved by imposing precedence orderings. Deleted-condition interactions can often be resolved in other ways as well—and thus, in general, they are more difficult to deal with than action-precedence interactions. However, there is a significant class of problems, including

certain kinds of automated manufacturing problems and certain kinds of scheduling problems, where action-precedence interactions are the only form of deleted-condition interactions. Examples of such problems appear later in this section.

2. Plans for different goals may sometimes contain some of the same actions. The *identical-action* interaction occurs when an action in one plan must be identical to an action in one of the other plans.

3. Sometimes, two different actions must occur at the same time. We call such an interaction a *simultaneous-action interaction.* This is different from the identical-action interaction, because these simultaneous actions are not identical. An example would be two robotic hands working together in order to pick up an object. As for identical-action interactions, we require that an action can occur simultaneously with at most one other action.

4. Let A be a set of actions $\{a_1, a_2, \ldots, a_n\}$. Then there may be a *merged action* $m(A)$ capable of accomplishing the effects of all actions in A. The cost of $m(A)$ could be either higher or lower than the sum of the costs of the other actions—but it is only useful to consider merging the actions in A if this will result in a lower total cost. Thus, although we allow the case where $\text{cost}(m(A)) \geq \sum_{a \in A} \text{cost}(a)$, we can ignore it for the purposes of planning. Thus, we only consider A to be mergeable if $\text{cost}(m(A)) < \sum_{a \in A} \text{cost}(a)$. Formally, we say that an *action-merging interaction* occurs among the actions in set A iff there is an action $m(A)$ such that

   (a) $\bigcup_{a \in A} \text{effects}^*(a) \subseteq \text{effects}(m(A))$, where $\text{effects}^*(a)$ for action a is the set of effects of a which are needed by some other actions in the same plan in which a is,

   (b) $\text{preconditions}(m(A)) \subseteq \bigcup_{a \in A} \text{preconditions}(a)$, and

   (c) $\text{cost}(m(A)) < \sum_{a \in A} \text{cost}(a)$.

   Notice that for a given set of actions, there may be more than one action $m(A)$ capable for achieving all of the effects of the actions while having a lower cost. Moreover, these merged actions can have different side-effects, which are effects other than those in $\bigcup_{a \in A} \text{effects}^*(a)$.

One way in which an action-merging interaction can occur is if the actions in A contain various sub-actions which cancel each other out, in which case the action $m(A)$ would correspond to the set of actions in A with these sub-actions removed.

Note that even though a set of actions may be mergeable, it may not always be possible to merge that set of actions in a given plan. For example, suppose a and a' are mergeable, but in the plan P, a must precede b and b must precede a' (see Fig. 6.1) . Then a and a' cannot be merged in P, because it would require b to precede itself.

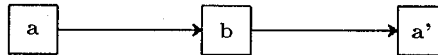


Figure 6.1: A plan containing actions that cannot be merged

Consider a set of plans $P_1, P_2, \ldots, P_g$ with a set of interactions of the types considered above. The action-precedence, identical-action and simultaneous-action interactions impose more precedence restrictions among the actions in the plan. These precedence restrictions, along with the original precedence arcs in the plans, determine whether the goals can be achieved by these plans. It is impossible to achieve the goals if the resulting global plan contains cycles of the actions. However, it the resulting plan is still a partially ordered set of actions, then the goals can be achieved together through the plans. In this case, we say that the plans can be combined, and define combine($\{P_1, P_2, \ldots, P_g\}$) to be the set of plans with the action-precedence, identical-action and simultaneous-action interactions handled.

The other kind of interaction (the action-merging interaction) places no constraint on how the plans might be combined, but instead allows possible modifications to the global plan once it has been created. Thus, the only kinds of interactions which might make it

impossible to combine a set of plans into a global plan are the action-precedence, identical-action, and simultaneous-action interactions. The problem of finding out whether or not a set of plans can be combined into a global plan we call the *multiple-goal plan existence problem.*

As an added complication, each goal G_i may have several alternate plans capable of achieving it, and thus there may be several different possible identities for the global plan for G. The least costly plan for G_i is not necessarily part of the least costly global plan, because some more costly plan for G_i may be mergeable in a better way with the plans for the other goals.

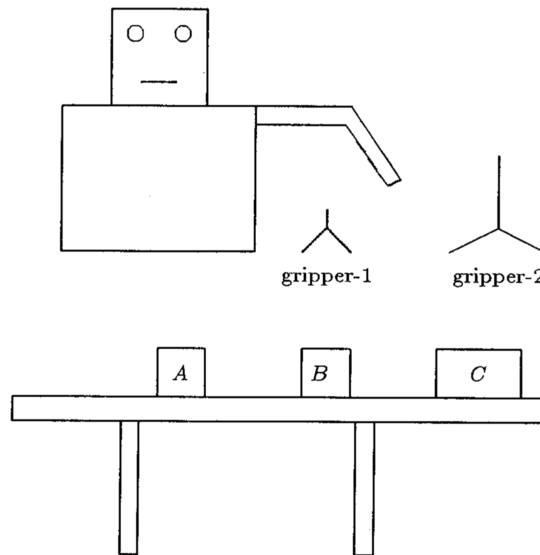


Figure 6.2: A blocks world problem where there are two different kinds of grippers

We define the *multiple-goal plan optimization problem* to be the problem of choosing which plan to use for each goal, and which actions to merge in these plans, so as to produce the least costly global plan for G.

The multiple-goal plan optimization problem occurs in a number of problem domains. Two examples are given below.

Example 6.1 *Consider a blocks-world problem in which the robot hand has several different grippers, for picking up several different types of blocks (Fig. 6.2). Only one gripper can be mounted on the hand at any given time. Suppose the action* pickup(B) *consists of mounting the appropriate gripper for block B and then picking up B, and the action* putdown(B) *consists of putting down B and removing the current gripper. Then the sequence of actions*

$$(\text{putdown}(A), \text{pickup}(B))$$

would consist of putting down A, removing the gripper for A, mounting the appropriate gripper for B, and picking up B. If A and B were the same kind of block, then these actions could be merged into an action which put down A and picked up B without changing grippers.

Suppose there are several stacks of blocks on a table, and the goal is to transform each stack into a desired goal stack. The transformation of each stack can be considered as an individual goal, and the collection of the goals constitutes a multiple goal planning problem. If the table is large enough, then moving the blocks on one stack does not interfere with any other stacks, and therefore, the only kind of interaction among the goals is the action-merging interaction described above.

Example 6.2 *Consider the automated manufacturing problem of drilling holes in a metal block. Several different kinds of hole-creation operations are available (twist-drilling, spade-drilling, gun-drilling, etc.), as well as several different kinds of hole-improvement operations (reaming, boring, grinding, etc.). Each time one switches to a different kind of operation or to a hole of a different diameter, one must put a different cutting tool into the drill. Suppose it is possible to order the operations so that one can work on holes of the same diameter at the same time using the same operation. Then these operations can be merged by omitting the task of changing the cutting tool. This problem is of practical significance (see [31]). (A related practical problem is discussed in [15].)*

For example, consider the problem of deciding how to make two holes Hole1 *and* Hole2 *in a metal block (see Fig. 6.3). Suppose the best plan for* Hole1 *is*

$$\text{spade-drill}(\text{Hole1}), \text{rough-bore}(\text{Hole1}), \tag{6.1}$$

and the best plan for Hole2 *is*

$$\text{twist-drill(Hole2), rough-bore(Hole2).} \tag{6.2}$$

If Plans 6.1 and 6.2 are combined naively, the result is

$$\text{spade-drill(Hole1), rough-bore(Hole1), twist-drill(Hole2), rough-bore(Hole2),}$$

which requires four tool changes. If the two holes have the same diameter, a more intelligent way to combine the plans is to do the two rough bore operations at the same time, saving a tool change:

$$\text{spade-drill(Hole1), twist-drill(Hole2), rough-bore(Hole1, Hole2).}$$

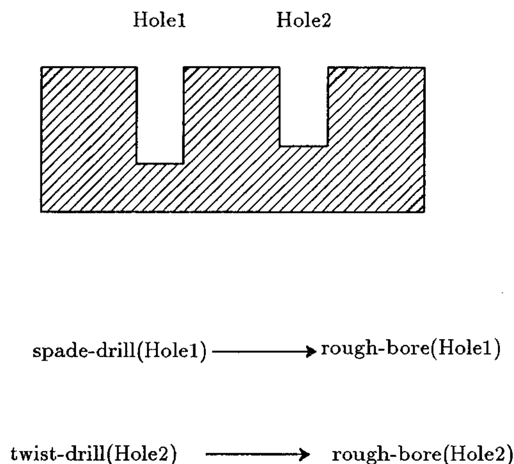


Figure 6.3: Merge two process plans for creating two machining holes

6.3 The Detection of Goal Interactions

In this section, we discuss how the interactions between the multiple goals can be found. It is assumed that g goals are to be solved, and there is one plan for each goal.

If a planner has no *a priori* knowledge about what kinds of goal interactions it is dealing with, then in general it is very costly to figure out what these interactions are. Take

deleted-condition conflicts as an example. Due to the undecidability of logical implications, it is not always possible to determine whether or not a proposition is denied by a group of others. Therefore, it is in general an unsolvable problem to find out a complete set of harmful goal interactions in a given set of plans. Moreover, even without the presence of axioms, the problem of determining the truth of a proposition in a set of plans in which the representation of actions are rich enough to represent effects dependent upon the input situation is NP-complete (see [7]). Therefore, even without axioms, the problem of finding out a complete set of harmful interactions is NP-hard.

However, the situation is not hopeless. First of all, when all the pre- and post-conditions of actions are simple first order logic propositions not dependent upon input situations, and when there are no additional axioms, the problem of finding out a set of deleted-condition conflicts between pairs of actions has a complexity that is quadratic as a function of the number of actions involved.

Even when there are external axioms, or there are actions with effects dependent upon input situations, it is possible that the propositions through which the goals interact are not dependent upon input situations, and not present in the axioms. This is the case with many problems in automated manufacturing domain. For example, a bore operation can have effects such as size(hole) = size(hole) + δ. This effect depends on the size of the hole before a bore action is applied, that is, it depends on the input situation. However, the size of one hole cannot be affected by the bore operations applied to other holes, as long as the holes are far away enough from each other. Therefore, the complexity of the planning algorithm is not increased because of this action representation.

Among the four kinds of goal interactions listed, the action-precedence interactions can be detected by comparing every pair of actions in the plans. If any precondition of one action a_i is deleted by the effects of another action a_j, then (a_i, a_j) should be recorded as an action-precedence interaction.

Identical-action and simultaneous-action interactions can be detected in terms of the domain knowledge available. They can also be detected via the constraints imposed upon the plans during planning for each individual goal. Action-merging interactions can be detected in terms of the representation of actions. For example, in example 6.2, a twist-drill operation is an compound action which can be decomposed into several subactions: mount(twist-drill) $\prec$ twist-drill(Hole) $\prec$ unmount(twist-drill). If two holes exist so

that the same twist-drill can be used for both, then the unmount(twist-drill) operation for the first hole can be canceled with the mount(twist-drill) operation for the second hole. The resultant overall plan then is of shorter length than without merging.

In the next two chapters, we consider two different cases of the multiple-goal plan optimization problem. The first case is when a single plan is generated for each goal. In this case, we show that the global plan optimization problem is NP-hard. However, a set of restrictions exists which defines a class of problems that is reasonably large and interesting, and can be solved in low-order polynomial time.

The second case is where more than one plan may be generated for each goal, making it necessary to select among the alternate plans for each goal to find an optimal global plan. This case is still NP-hard even with the restrictions, but there is a heuristic approach which works well in practice.

Chapter 7

One Plan For Each Goal

Planning is often so difficult that most planning systems stop once they have found a single plan for each goal, without trying to find other plans as well. This section discusses the multiple-goal plan optimization problem in the case where only one plan is available for each goal.

7.1 Complexity

To examine the computational complexity of the problem, consider the special case in which the following conditions hold: for each goal G_i, the plan P_i for G_i is a linear sequence of actions, each action has a cost of 1, and there are no interactions among the plans except for action-merging interactions.

In this special case, all interactions that might prevent the plans from being combined into a global plan are disallowed. Thus, at least one global plan is guaranteed to exist: $\Pi = \bigcup_{i=1}^{g} P_i$. However, Π may not be an optimal plan, because it may be possible to merge some of its actions. In fact, Π may contain several different sets of mergeable actions, and merging some of them may preclude merging others. Different choices of which sets to

merge may result in plans of different cost. For example, consider two plans P_1 and P_2 in Fig. 7.1 (a). Assume that actions which name start with the same characters can be merged. Fig. 7.1 (b) and (c) show two different ways the two plans can be merged.

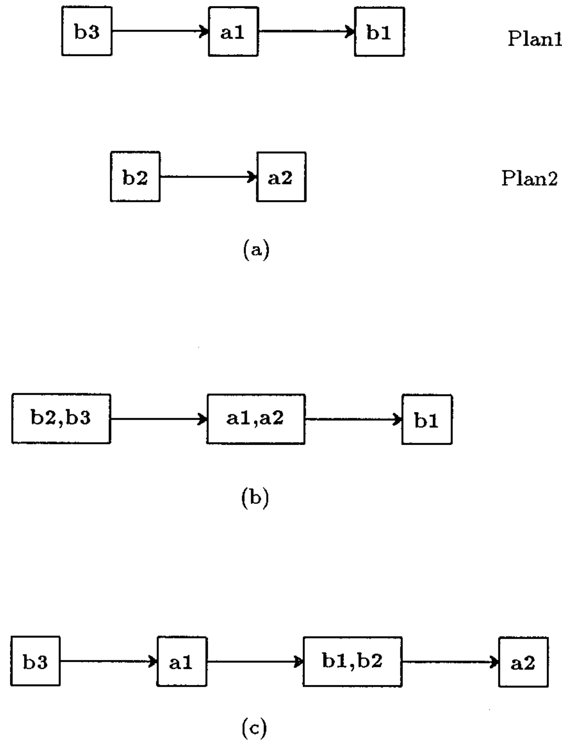


Figure 7.1: Plans which can be merged in different ways.

Since this special case requires that each P_i be a linear sequence of actions and each action have the same cost, the problem of finding the best global plan is equivalent to the problem of finding the shortest common supersequence (SCS) of n sequences. This problem has been shown to be NP-hard [27]. Since the special case is NP-hard, it follows that the multiple-goal plan optimization problem is NP-hard.

7.2 Plan Existence

The last section showed that the multiple-goal plan optimization problem with one plan per goal is NP-hard, by showing that a special case of that problem is NP-hard. In this

section, we return to the original problem with one plan per goal, in which the plans can interact with each other via action-precedence, action-merging, identical-action and action-merging interactions.

One way to handle a NP-hard problem is to simplify it either by relaxing the criteria for what constitutes a valid solution, or by imposing restrictions on what problem instances will be considered. For the current problem, one way to do this is to look not for the optimal global plan, but for any global plan that works—in other words, to solve the multiple-goal plan existence problem rather than the multiple-goal plan optimization problem. Whether a global plan exists is independent of whether there are any action-merging interactions, so for the multiple-goal plan existence problem we can ignore all action-merging interactions completely.

In particular, suppose that we are given the following:

1. A set of plans $S = P_1, P_2, \ldots, P_g$ containing one plan P_i for each goal G_i. Let n be the total number of actions in S.
2. A list of interactions among the actions in the plans (for example, members of this list might be statements such as "action a in plan P_i must precede action b in plan P_j"). Let i be the total number of interactions in this list (note that $i = O(n^2)$).

If we do not do any merging of actions, the global plan is just the set of individual plans in S, plus certain ordering constraints upon the actions which are required for handling the interactions. If the resultant global plan exists, this *combined* plan, combine(S), can be produced by the following algorithm:

Algorithm 7.2

1. For each plan P in S, create a graph representing P as an adjacency list. Also, create a sorted linear index of the actions in the plans. This step can be done in time $O(n^2)$.

2. For each action-precedence interaction in the interaction list, modify the graph by creating a precedence arc between the actions named in the interaction. For each simultaneous-action interaction in the interaction list, create a simultaneous-action arc between the actions named in the interaction. For each identical-action interaction in the interaction list, combine the actions named in the interaction into a

single action. If this step is done by sorting the interaction list and then checking it against the index of actions, it can be done in time $O(i \log i + (i+n)n) = O(n^3)$.

3. Check to see whether the graph still represents a consistent partial ordering (this can be done in time $O(n^2)$ using a topological sorting algorithm [19], with a straightforward extension to handle the simultaneous-action interactions). If it does not, then exit with failure (no global plan exists for this problem).

Algorithm 7.2 produces the combined plan combine(S) if it exists, in the case where there is one plan for each goal G_i. The total time required is $O(n^3)$, where n is the total number of actions in the plans.

7.3 Plan Optimality with Restrictions

Section 7.2 dispensed with action-merging interactions by considering the multiple-goal plan existence problem rather than the multiple-goal plan optimization problem. In this section, we consider how to solve the multiple-goal plan optimization problem by imposing restrictions on how the actions can interact with each other. As in the previous section, we assume that one plan is available for each goal.

According to our definition, merging a set of actions corresponds to replacing them by another action with less cost. We have demonstrated that in general the plan optimization problem is NP-hard. One of the reasons for this complexity is that merging one set of actions may preclude merging some other sets of actions. In order to find out which set of actions gives the best choice for merging, the results of all possible ways of merging should be compared.

Another source of complication is that merged actions may inflict new conflicts in the plans. Suppose it is decided to merge the actions in the set $\{a_1, a_2, \ldots, a_k\}$ into another action a_m, such that a_m can achieve all of the effects of the a_is. In addition to the effects of a_i, $i = 1, \ldots, k$, a_m may also assert some other effects, say $\neg q$, where q is part of the preconditions for some other action a_o. Effects like this can be called *harmful side-effects.* In this case, merging the actions in set $\{a_1, a_2, \ldots, a_k\}$ may create an action-precedence interaction, depending on the relative positions of a_m and a_o. If a_m and a_o are unordered in the multiple goal plan, then it is necessary to introduce precedence link from a_o to a_m,

after the merging is done. However, if a_m is before a_o in the plan, and no other actions can restore the precondition of a_o, q, then merging $\{a_1, a_2, \ldots, a_k\}$ will produce an invalid multiple-goal plan.

To make our planning task easier, restrictions will be imposed on how the actions in the plans can interact via action-merging interactions:

Restriction 7.1 *Merging does not have harmful side-effects.*

Restriction 7.2 *If S is a set of plans, then the set of all actions in S may be partitioned into equivalence classes of actions $E_1, E_2, \ldots, E_p$, such that sets of actions A and B are mergeable if and only if A and B are subsets of the same equivalence class.*

Restrictions 7.1 and 7.2 are reasonable for a number of problems (for example, it is already satisfied in the Examples 6.1 and 6.2 discussed previously). These restrictions are still not restrictive to reduce the complexity of the problems to polynomial. However, they reduce the total size of the state space by reducing the base of the complexity formula. In the following, we will present a branch-and-bound algorithm for solving the multiple goal plan optimization problem where these restrictions are satisfied.

7.3.1 A Branch-and-Bound Algorithm

Consider domains where restrictions 7.1 and 7.2 are satisfied. Let P be combine($\{P_1, P_2, \ldots, P_g\}$), where each P_i is a plan for achieving a goal G_i. P can be computed using Algorithm 7.2. Let M be a set $\{A_1, A_2, \ldots, A_n\}$, where each set A_i is a mergeable set of actions in the plans. Each A_i is called a *merge-set.* For the simplicity of the algorithm description, assume that each merge-set is a pair of actions that belong to the same equivalence class. It is possible that merging the actions in set A_i may preclude merging the actions in set A_j. The goal of the search algorithm is to select a set of merge-sets so that after merging the actions in each set, the global plan will be the least-cost plan which is free of cycles.

The state space for the search algorithm is a decision tree based on the set of merge-sets M (see Fig. 7.2). Each state S in the tree contains a set of chosen merge-sets $C(S)$, the remaining set of merge-sets $M(S)$, and a set $U(S)$ of action pairs which are not mergeable.

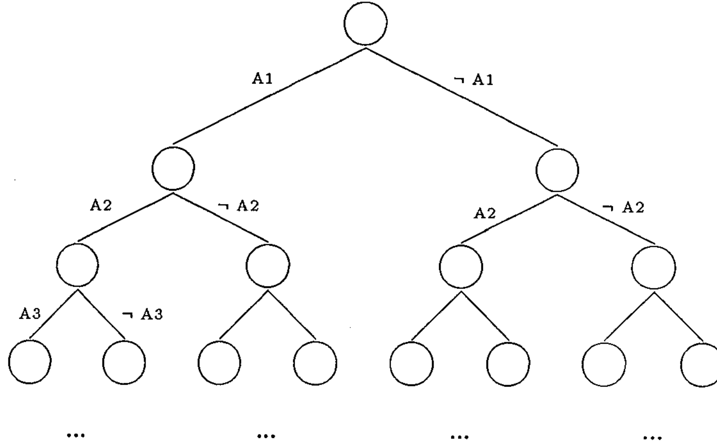


Figure 7.2: State space tree for the branch and bound algorithm.

The root of the state space search tree contains an empty set of chosen merge-sets. In the search tree, each state S has two children. The first child corresponds to the decision to include the next merge-set A_i of $M(S)$ into $C(S)$, and the second child to exclude this pair from the final selection. A final state is a state S such that $M(S)$ is empty, and a goal state is a final state with the least cost.

One important question to ask about a search space is whether it is complete. A complete state space always includes a solution for a problem instance as one of its state, whenever such a solution exists. Clearly, the state space described above is complete, because it considers every possible way the actions in the plans can be merged.

Another important issue for branch-and-bound search algorithms is whether an informed lower bound function can be found for each state. Let $L(S)$ be the lower bound for state S. It can be constructed as follows.

As described above, associated with each state S is the following information:

M(S): a set of remaining merge-sets. For the root R, $M(R) = M$, where M is the original set of merge sets.

U(S): a set of pairs of actions that belong to the same equivalence class, but cannot be merged because of precedence constraints. Initially, $U(S)$ is the set of pairs of actions

(a_i, a_j), such that there is a precedence arc from a_i to a_j. The precedence arc may be imposed because of the initial structure of the plan, or one of the action-precedence, simultaneous-action or identical-action interactions.

C(S): the set of chosen merge-sets from the root down to the current state. If $A_i, A_j \in C(S)$ such that $A_i \bigcap A_j \neq \phi$, then according to Restriction 7.2, $A_i \cup A_j$ is a new merge-set.

P(S): the plan resulting from merging the merge-sets in $C(S)$.

Let $U_i(S)$ be the subset of U for the ith equivalence class. The lower bound cost function $L(S)$ can be computed by summing up the lower bound $L_i(S)$ for each individual equivalence class i, since no actions belonging to different equivalence classes can be merged. Therefore,

$$L(S) = \sum_i L_i(S).$$

The computation of $L_i(S)$ depends on whether the subset of merge-sets belonging to this class i is empty in set $M(S)$. If it is empty, then the state S is a leaf node of the search tree, and the exact cost of the part of the global plan composed of actions in this class i can be obtained as follows:

$$L_i(S) = \sum_{A_i \in C(S)} \text{cost}(\text{merge}(A_i)),$$

where each A_i is a merge-set consisting of actions in class i.

If $M(S)$ is not empty, then S is a non-terminal node. In this case, the lower bound cost function can be computed in terms of the set of unmergeable action pairs in $U(S)$. Let $U_i(S)$ be the subset of $U(S)$ containing only action pairs in class i. Let $D_i(S)$ be a clique defined by the set $U_i(S)$. $D_i(S)$ is a set of actions such that $a, a' \in D_i(S)$ if and only if $(a, a') \in U_i(S)$. Then

$$L_i(S) = \max \left\{ \sum_{a \in D_i(S)} \text{cost}(a),\ \text{cost}(\text{merge}(\{a \mid a \in \text{ equivalence class } i\})) \right\}. \quad (7.1)$$

A clique D_i in $U_i(S)$ is computed using a greedy algorithm. Given a set of unmergeable pairs of actions $U_i(S)$, this greedy algorithm selects a set $D_i(S)$ such that no two actions in $D_i(S)$ can be merged due to the precedence constraints.

The admissibility of this lower bound function can be proved by showing that, for each equivalence class i, $L_i(S)$ is a lower bound. That is, we have to show $L_i(S) \leq \text{cost}_i(\text{merge}(S'))$, where $\text{cost}_i(\text{merge}(S'))$ is the cost of the actions belonging to equivalence class i, in any final state S' below the current state S.

Since in the best situation, all actions of type i can be merged, the cost of S' contributed by the actions of in class i should be at least

$$\text{cost}(\text{merge}(\{a \mid a \in \text{ equivalence class } i\}))$$

On the other hand, since the actions in $D_i(S)$ cannot be merged, they cannot be merged in state S' either. Therefore, the actions belonging to equivalence i in S' must be divisible to at least $|D_i(S)|$ parts so that each part contains the actions that can be merged, and no two sets have actions that will be merged. For each such set d_{ij}, $\text{cost}(d_{ij})$ is greater than the cost of any individual action. Therefore,

$$\text{cost}_i(\text{merge}(S')) \geq \sum_{a \in D_i(S)} \text{cost}(a).$$

Since neither of the formulae in Equation 7.1 is greater than $\text{cost}_i(\text{merge}(S'))$, $L_i(S)$ must be a lower bound function.

Let S be a non-final state. As we discussed above, a child of S can be obtained by removing the next merge-set A_i from $M(S)$. The first child of S, $\text{first}(S)$, can be created by including A_i in $C(S)$, while the second child, $\text{second}(S)$ exludes A_i from $C(S)$. In the following, we consider the complexity of producing only the first child from state S. The complexity of creating the second child can be found in a similar way.

For $\text{first}(S)$, the following has to be done:

1. Check the feasibility of the plan in $\text{first}(S)$. This can be done in time $O(n^2)$ using the topological sort algorithm, where n is the total number of actions in the plan. In addition, tests have to be made to see whether certain action-pairs in $U(S)$ are merged as a result of merging the actions in A_i. If so, then the state $\text{first}(S)$ is infeasible. Overall, this step takes $O(n^2)$ in time complexity.

2. Update $M(S)$. After the actions in A_k are merged, a number of other actions may no longer be merged. Suppose $A_k = (a_i, a_j)$. If an action a' precedes a_i, and a'' is preceded by a_j, then even though a' and a'' belong to the same equivalence class,

they cannot be merged. To find out all the pairs like this, one can check if an arc exists linking any pairs of the actions in set $M(S)$. If (a', a'') is a pair removed from $M(S)$ in this test, it should be inserted into $U(\text{first}(S))$. This step takes time $O(n^2)$.

3. Compute $L(\text{first}(S))$. This step can be done with the greedy algorithm above. Since this algorithm takes time linear in the number of unmergeable pairs in $U(S')$, and the set $U(S')$ has a size of $O(n^2)$, the worst case time complexity for this step is $O(n^2)$.

Therefore, the worst case time complexity for processing each state is $O(n^2)$.

7.3.2 Empirical Results

The total size of the state space can be estimated from the definition of the search algorithm. The branching factor of the search tree is 2, and the depth of the search tree is the total number of mergeable pairs of actions. Therefore, the size of the whole search tree is $2^{|M|}$. That is, reducing the total number of mergeable sets of actions can decrease the size of the search tree by an exponential amount. Furthermore, the amount of processing at each state is also proportional to the number of merge-sets. Thus one other criterion for identifying domains with limited action-merging interactions is to check whether the total number of merge-sets is small. If this number is small, the branch-and-bound algorithm will be efficient.

A preliminary experiment has been done with the branch-and-bound algorithm. In this experiment, the cost of each action is assumed to be 1, and the cost of a plan is the total number of actions in it. This assumption is of practical importance. For example, this cost definition is useful for minimizing the total number of tool changes in the final plan, when each equivalence class corresponds to a different tool.

The number of states explored v.s. number of mergeable pairs is shown in Fig. 7.3. The number of states expanded by the algorithm shows quadratic behavior as a function of the number of merge-sets, when the number of merge-sets is less than 40.

7.4 Partial Ordering Restriction

This section considers one more restriction which makes it feasible to look for an optimal global plan with a polynomial time algorithm, rather than a heuristic search algorithm. Here we assume that Restrictions 7.1 and 7.2 are satisfied. In addition, we require that the following restriction is also satisfied:

Restriction 7.3 *If* combine(S) *exists, then it defines a partial order over the equivalence classes defined in Restriction 7.2; i.e., if* E_i *and* E_j *are two distinct equivalence classes and if* combine(S) *requires that some action in* E_i *occur before some action in* E_j*, then* combine(S) *cannot require that some action in* E_j *occur before some action in* E_i*. (This does not rule out the possibility of an action in* E_i *occurring immediately before another action in* E_i*; in such a case, the two actions can be merged.)*

Intuitively, Restriction 7.3 requires that merging one set of actions in an equivalence class does not preclude the possibility of merging other actions in the plans. For example, the plans in Fig. 7.1 do not satisfy Restriction 7.3, since merging actions $b1$ and $b2$ will preclude merging actions $a1$ and $a2$.

Although this restriction is more limiting, it still allows many interesting problems. For instance, in Example 6.1 it would be satisfied if the grippers had to be used in a certain order; and in Example 6.2 it is satisfied since there exists a common sense ordering of the machining operations (e.g., don't twist-drill a hole after it has been bored), which is quite natural to use for this problem.

Suppose Restrictions 7.1, 7.2 and 7.3 are all satisfied, and suppose we are given a set of plans S and a list of interactions, as was done in Section 7.2. If a global plan exists, then Algorithm 7.2 will produce the global plan combine(S). However, by merging some of the actions in combine(S), it may be possible to find other less costly plans. The following algorithm will find the least costly of these plans.

Algorithm 7.4

1. Use Algorithm 7.2 to produce a digraph representing the combined plan combine(S). This can be done in time $O(n^3)$. If Algorithm 7.2 succeeds, then continue; otherwise, exit with failure.

2. For each equivalence class E_i of actions in combine(S), merge all of the actions in E_i. This produces a digraph in which each class of action occurs only once (e.g., see Fig. 7.4). From Restriction 7.3, it follows that this is a consistent plan; we call this plan merge(combine(S)). From Restriction 7.2 and the definition of mergeability, it can be proved by induction that this is the least costly plan which can be found by combining and merging actions in S. Merging the classes will, at worst, require redirecting all of the arcs in the digraph—and this can be done in time $O(n^2)$.

In the case where there is one plan for each goal G_i, Algorithm 7.4 produces an optimal way to combine and merge these plans, if it is possible to combine them at all. The total time required is $O(n^3)$, where n is the total number of actions in the plans.

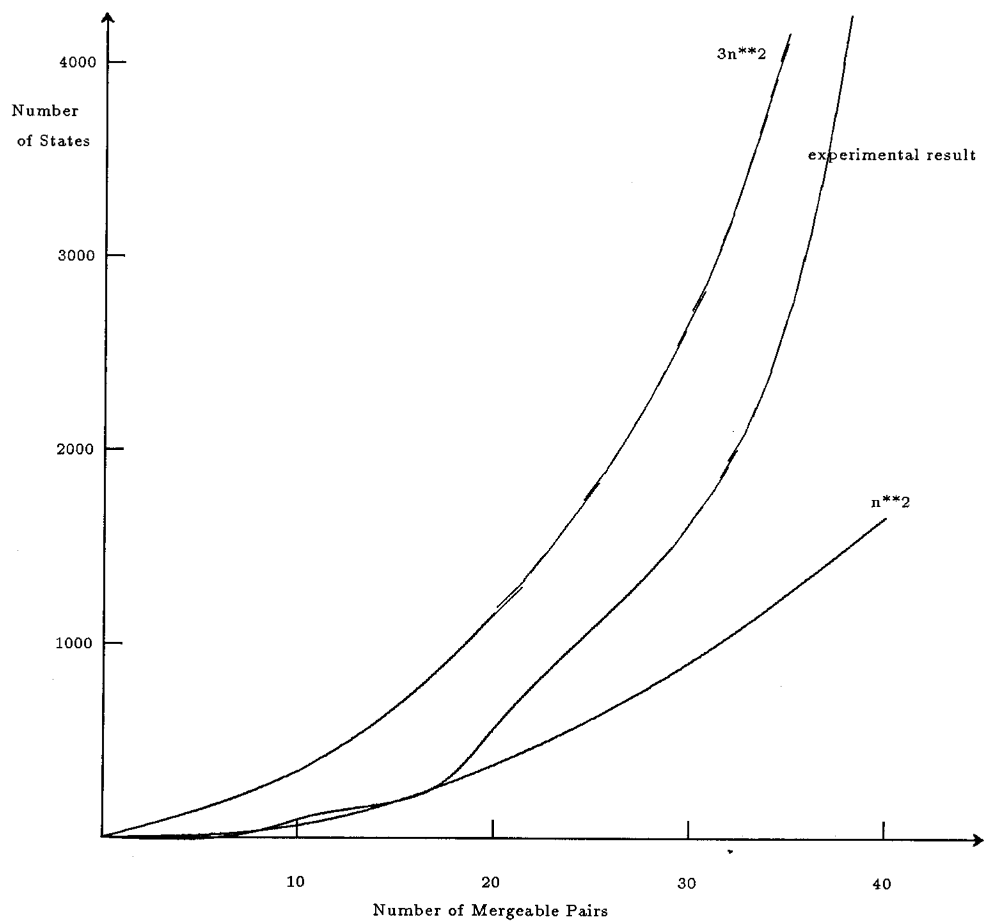


Figure 7.3: Experimental results for merging a set of plans

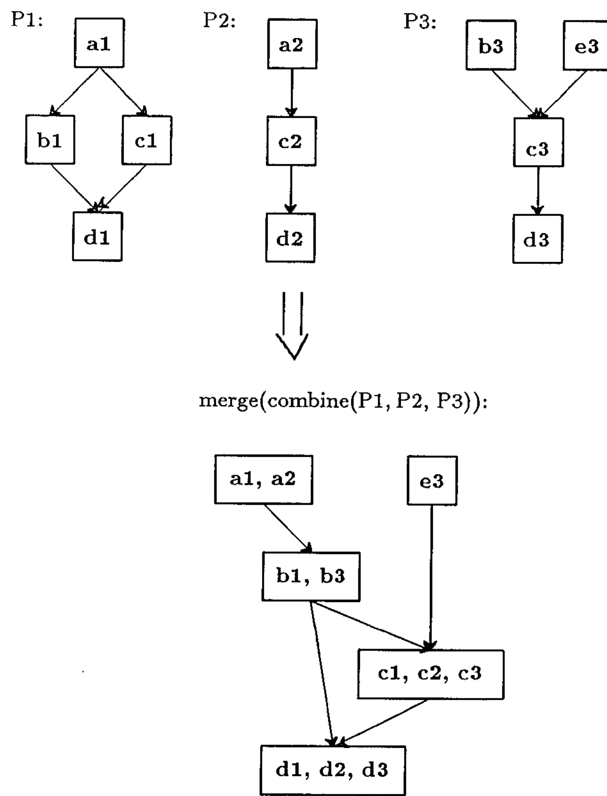


Figure 7.4: Finding an optimal global plan by merging the plans for various goals.

Chapter 8

More Than One Plan For Each Goal

For some multiple goal planning problems, it is reasonable to expect that more than one plan may be found for each goal. (For example, this is done by the SIPS planning system for the manufacturing problem discussed in Example 6.2 [31]). Finding more than one plan for each goal is computationally more complex than finding just one plan for each goal, but it is useful because it can lead to better global plans.

To see this, consider once again the process planning example described in Example 6.2 of Chapter 6. Suppose in addition to the plans given in Example 6.2, Hole2 can also be created (at slightly higher cost) using the plan

$$\text{spade-drill(Hole2)}, \text{rough-bore(Hole2)}. \tag{8.1}$$

(see Fig. 8.1) Even though this plan costs slightly more than plan 6.2, it results in a better overall plan when combined with plan 6.1, because one additional tool change can be saved:

$$\text{spade-drill(Hole1, Hole2)}, \text{rough-bore(Hole1, Hole2)}.$$

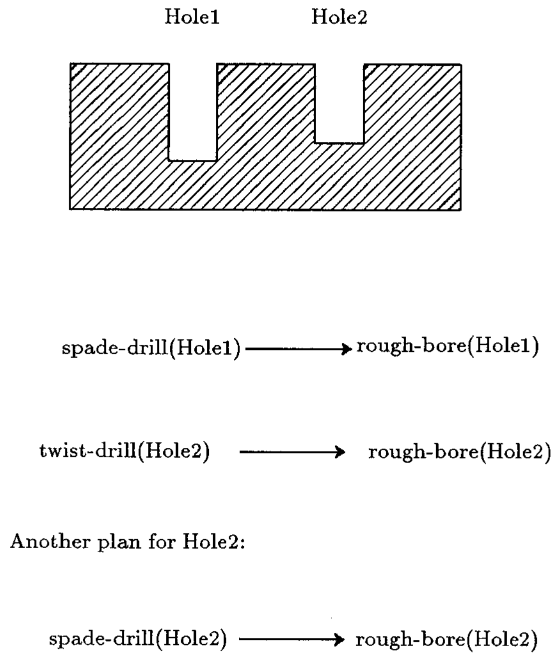


Figure 8.1: More than one process plan is available for Hole2.

In the following, we first show that the multiple goal plan existence problem is NP-hard when there is more than one plan for each goal. Then a branch-and-bound algorithm will be presented for solving the problem.

8.1 Complexity

If more than one plan is available for each G_i, then there may be several different possible identities for the set S discussed in the previous chapter, and it may be necessary to try several different possibilities for S in order to find one for which combine(S) exists. This problem is NP-hard, even with Restrictions 7.1, 7.2 and 7.3; this is proved in Appendix B by reducing CNF-satisfiability to it.

Since the multiple-goal plan existence problem is a special case of the multiple-goal plan optimization problem, the above result means that the multiple-goal plan optimization

problem is also NP-hard.

Polynomial-time solutions do exist for a number of special cases of the multiple-goal plan existence and optimization problems. One such special case was the one discussed in Section 7.1, in which the number of plans for each goal was taken to be 1. Another special case is the case where the number of different equivalence classes of actions is less than 3, and each action has the same cost. In this case, if no conflicting constraints are allowed to exist, the multiple-goal plan optimization problem can be solved in polynomial time, even if Restriction 7.3 is lifted. For example, this would occur in example 6.1 if there were only two different grippers and two different types of blocks.

8.2 A Heuristic Algorithm

Although the general case of the multiple-goal plan optimization problem is NP-hard, there is a heuristic approach that performs well in practice on this problem. The approach is to formulate the problem as a state-space search and solve it using a best-first branch-and-bound algorithm. In the following, we assume that Restrictions 7.1, 7.2 and 7.3 hold for the problem domain.

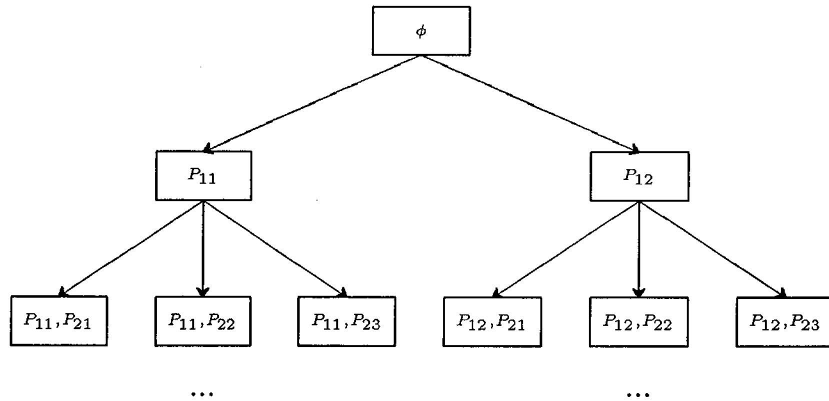


Figure 8.2: An example state space. Here P_{ij} is the jth alternative plan for the goal G_i.

Suppose we are given the following: (1) for each goal G_i, a set of plans T_i containing one or more plans for G_i, and (2) a list of the interactions among the actions in all of the plans. In the state-space search, the state space is a tree. Each state is a set of plans; it

contains one plan for each of the first i goals for some i. The initial state is the empty set (i.e., $i = 0$). If S is a state containing plans for the goals $G_1, G_2, \ldots, G_i$, then an immediate successor of S is any set $S \cup \{P\}$ such that P is a plan for G_{i+1}. A goal state is any state in which plans have been chosen for all of the goals $G_1, G_2, \ldots, G_g$. The cost of a state S is the cost of the plan obtained by applying Algorithm 7.4 to S; i.e.,

$$\text{cost}(S) = \text{cost}(\text{merge}(\text{combine}(S))).$$

Fig. 8.2 displays part of an example state space.

The search algorithm will be referred to as Algorithm 8.2. As the algorithms in the previous chapter, this algorithm is a best-first branch-and-bound search, which uses a lower bound function L to order the members of the list of alternatives being considered. Except for the use of U for pruning, this algorithm can also be thought of as a version of the A* search procedure, with $h(S) = L(S) - \text{cost}(S)$ as the heuristic function.

In the search algorithm, pruning may be done by computing an upper bound on the cost of the best global plan. For each G_i, let best(G_i) be the plan for G_i of least cost. The plans best(G_i), $i = 1, 2, \ldots, g$, may or may not be combined depending on the interactions among them. This can be determined using Algorithm 7.2 in time $O(m^3)$, where m is the total number of actions in $\{\text{best}(G_1), \text{best}(G_2), \ldots, \text{best}(G_g)\}$. If they cannot be combined, U can be initialized to be infinity. On the other hand, if combine$(\{\text{best}(G_1), \text{best}(G_2), \ldots, \text{best}(G_g)\})$ exists, then the upper bound is

$$U = \text{cost}(\text{merge}(\text{combine}(\{\text{best}(G_1), \text{best}(G_2), \ldots, \text{best}(G_g)\}))). \tag{8.2}$$

During the search, any state S such that $L(S)$ is greater than U is discarded.

If $L(S)$ is a lower bound on the costs of all successors of S that are goal states, then L is admissible, in the sense that Algorithm 8.2 will be guaranteed to return the optimal solution. We now discuss various possible functions to use for L. To do this, we temporarily assume the following property: that merging plans for two different goals always results in a plan at least as expensive as either of the two original plans. In other words, if P and Q are plans for two distinct goals, then

$$\text{cost}(\text{merge}(\text{combine}(P, Q))) \geq \max(\text{cost}(\text{merge}(P)), \text{cost}(\text{merge}(Q))). \tag{8.3}$$

We will later discuss what happens when this property is not satisfied.

If Eq. 8.3 is satisfied, then clearly $L_0(S) = \text{cost}(S)$ is a lower bound on the cost of any successor of S (this would correspond to using $h \equiv 0$ in the A* search algorithm). However, a better lower bound can be found as follows. Suppose S contains plans for $G_1, \ldots, G_i$. For each $j > i$, let $P^*(S,j)$ be the plan P for G_j which minimizes cost(merge(combine($S \cup \{P\}$))). Let

$$L_1(S) = \max_{j>i} \text{cost}(\text{merge}(\text{combine}(S \cup \{P^*(S,j)\}))). \tag{8.4}$$

To show that L_1 is admissible, let S be any state, and let S^* be any descendant of S that is a goal state. We must show that $L_1(S) \leq \text{cost}(S^*)$. From Eq. 8.4, there is some $j > i$ such that

$$L_1(S) = \text{cost}(\text{merge}(\text{combine}(S \cup \{P^*(S,j)\}))). \tag{8.5}$$

But S^* contains some plan P_j for the goal G_j, and from the definition of $P^*(S,j)$,

$$\text{cost}(\text{merge}(\text{combine}(S \cup \{P^*(S,j\}))) \leq \text{cost}(\text{merge}(\text{combine}(S \cup \{P_j\}))). \tag{8.6}$$

Thus, by repeated application of Eq. 8.3,

$$L_1(S) \leq \text{cost}(\text{merge}(\text{combine}(S^*))), \tag{8.7}$$

so L_1 is admissible.

One way to find $P^*(S,j)$ is to compute merge(combine($S \cup \{P\}$)) for each plan $P \in T_j$ using Algorithm 7.4, and then select the plan P that yields the minimum cost. If P is a plan, let $A(P)$ be the set of actions in P. If the above approach is used to compute $L_1(S)$, the time required is

$$\sum_{j=i+1}^{g} \sum_{P \in T_j} O(|A(S)| + |A(P)|)^3. \tag{8.8}$$

During this computation, it may be realized that some P cannot be combined with S, due to conflicting constraints. In that case, the plan P can be removed from further consideration, simplifying the computation of L_1 on successors of S. Also, if none of the plans in T_j can be combined with S, then there is no possible way that S can be extended into a complete global plan, so search can be discontinued at S.

By sacrificing the quality of the lower bound somewhat, we can compute it more efficiently. We associate with each state S some sets $H_1(S), H_2(S), \ldots, H_g(S)$, which are computed as follows. For the initial state ($S = \emptyset$), for $j = 1, 2, \ldots, g$,

$$H_j(S) = \{A'(P) | P \text{ is a plan for } G_j\}, \tag{8.9}$$

where $A'(P)$ is merge($A(P)$). That is, $A'(P)$ is the set of actions in the plan resulting from merging all the actions in P.

Let S be any state at level $i-1$, and let S' be the state formed by including a plan P_i for the goal G_i. Then, for $j = i+1, \ldots, g$,

$$H_j(S') = \{Q' | Q \in H_j(S)\}, \tag{8.10}$$

where Q' is Q minus each action which falls into the same equivalence class as some action in P_i. We now define

$$L_2(S') = \text{cost}(\text{merge}(\text{combine}(S')) + \max_{j=i+1}^{g} \min\{\text{cost}(Q) | Q \in H_j(S')\}, \tag{8.11}$$

where cost(Q) is the sum of the costs of the actions in Q, and where the min of an empty set is taken to be 0.

We now show that L_2 is admissible. Let S' be defined as above. From Eq. 8.4, for $k = i+1, \ldots, g$,

$$L_1(S') \geq \text{cost}(\text{merge}(\text{combine}(S' \cup \{P^*(S', k)\}))). \tag{8.12}$$

For each k, partition the actions in $A(P^*(S', k))$ into two sets, A_1 and A_2. A_1 contains all the actions in $A(P^*(S', k))$ that fall into the same equivalence class as some action in S', while A_2 contains the rest in $A(P^*(S', k))$. From Eq. 8.4,

$$\text{cost}(\text{merge}(\text{combine}(S' \cup \{P^*(S', k)\}))) \geq \text{cost}(\text{merge}(\text{combine}(S' \cup \{A_2\}))). \tag{8.13}$$

Since A_2 contains actions that cannot be merged with the actions in S', and since combining the actions in a plan does not change the cost of the plan,

$$\text{cost}(\text{merge}(\text{combine}(S' \cup \{A_2\}))) \geq \tag{8.14}$$

$$\text{cost}(\text{merge}(\text{combine}(S'))) + \text{cost}(\text{merge}(\text{combine}(\{A_2\}))). \tag{8.15}$$

From the definition of Q (in Eq. 8.10),

$$\text{cost}(\text{merge}(\text{combine}(\{A_2\}))) \geq \min\{\text{cost}(Q) | Q \in H_k(S')\}. \tag{8.16}$$

To sum up, for $k = i+1, \ldots, g$,

$$L_1(S') \geq \text{cost}(\text{merge}(\text{combine}(S')) + \min\{\text{cost}(Q) | Q \in H_k(S')\}. \tag{8.17}$$

Therefore,

$$L_1(S') \geq \text{cost}(\text{merge}(\text{combine}(S')) + \max_{j=i+1}^{g} \min\{\text{cost}(Q)|Q \in H_j(S')\} \quad (8.18)$$
$$= L_2(S'). \quad (8.19)$$

Thus, from Eq. 8.18 and the admissibility of L_1, L_2 is also admissble.

Computing $L_2(S)$ takes time

$$\sum_{j=i+1}^{g} \sum_{V \in H_j(S)} O(|V| + |A(P_i)|) \quad (8.20)$$

This is much less than the time required for computing $L_1(S)$, for several reasons:

1. Comparing the expression $|V|+|A(P_i)|$ in Eq. (4.11) to the expression $|A(S)|+|A(P)|$ in Eq. (4.7), $|A(P_i)|$ is about the same as $|A(P)|$ assuming that the size of the plans are not very different, but $|V|$ is much smaller than $|A(S)|$. Furthermore, the expression $|A(S)| + |A(P)|$ is cubed in Eq. (4.7), and the expression $|V| + |A(P_i)|$ is not raised to a power in Eq. (4.11).

2. Initially, T_j and H_j have the same number of elements, which is the number of alternate plans for goal G_j. For increasing values of i, the size of $H_j(S)$ decreases, so that fewer elements are summed in Eq. (4.11). But in Eq. (4.7), the size of T_j is independent of i.

L_2 is usually less informed than L_1, because it only takes into account the actions which are not in the same equivalence classes as any actions in S'. However, if all actions have the same cost, then $L_2 = L_1$.

Some problems (e.g., example 6.1) satisfy the restriction given in Eq. (4.2), and others do not. If Eq. (4.2) is not satisfied, then there will be some states S such that $L_0(S)$, $L_1(S)$, and $L_2(S)$ are not lower bounds on S. For most reasonable planning problems, S can be shown to be dominated by states on other paths in the search tree, in which case Algorithm 8.2 is still guaranteed to return a least-cost solution (for a mathematical analysis of such cases, see [34]). In other cases, such S might not be dominated by states on other paths, in which case Algorithm 8.2 might not return the optimal solution—but in this case, Algorithm 8.2 will still return results that are close to optimal (for a mathematical analysis of such cases, see [16]).

Plan	Action_1	Cost	Action_2	Cost	Action_3	Cost	Total
P_{11}	a	20	b	15	c	5	40
P_{12}	d	35	e	10			45
P_{21}	a'	20	f	10	g	5	35
P_{22}	h	10	b'	15	i	10	35
P_{23}	j	30	e'	10			40

Table 8.1: Plans with their actions and costs.

Consider the following example. Suppose two goals G_1 and G_2 are given along with their plans: $P_{11}, P_{12}, P_{21}, P_{22}, P_{23}$, where P_{ij} is the jth alternate plan for goal G_i. Table 8.1 lists the costs for the actions involved in each plan. Each plan is a linear sequence of actions, so that an action under Action_i precedes an action under Action_j if $i < j$. Suppose that the only kind of goal interaction in this example is action-merging interaction. Also assume that for all actions x, x and x' belong to the same equivalence class. Moreover, $\text{merge}(x, x') = x$.

The search space for this example is shown in Fig. 8.3. The values of the heuristic functions L_1 and L_2 applied to the states S_i are shown below. Notice that the upper bound U can be computed by merging the plans P_{11} and P_{21}, yielding

$$U = 55.$$

If the heuristic function L_1 is used, then the following computation is done.

$L_1(S_1)$:

$$\text{merge}(\text{combine}(\{P_{11}, P_{21}\})) = \sum_{x \in \sigma} \text{cost}(x) = 55,$$

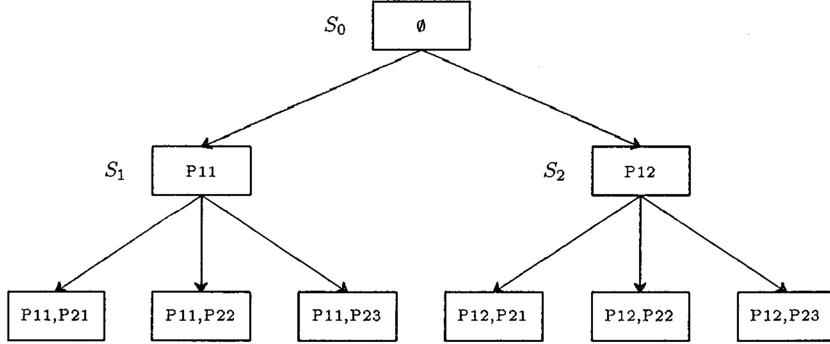


Figure 8.3: The search space for the example.

where $\sigma = \{a, b, c, f, g\}$;

$$\text{merge}(\text{combine}(\{P_{11}, P_{22}\})) = \sum_{x \in \sigma} \text{cost}(x) = 60,$$

where $\sigma = \{a, b, c, h, i\}$;

$$\text{merge}(\text{combine}(\{P_{11}, P_{23}\})) = \sum_{x \in \sigma} \text{cost}(x) = 80,$$

where $\sigma = \{a, b, c, j, e\}$.

Therefore, $L_1(S_1) = 55$.

$L_1(S_2)$:

$$\text{merge}(\text{combine}(\{P_{12}, P_{21}\})) = \sum_{x \in \sigma} \text{cost}(x) = 80,$$

where $\sigma = \{d, e, a, f, g\}$;

$$\text{merge}(\text{combine}(\{P_{12}, P_{22}\})) = \sum_{x \in \sigma} \text{cost}(x) = 80,$$

where $\sigma = \{d, e, h, b, i\}$;

$$\text{merge}(\text{combine}(\{P_{12}, P_{23}\})) = \sum_{x \in \sigma} \text{cost}(x) = 75,$$

where $\sigma = \{d, j, e\}$.

Therefore, $L_1(S_2) = 75$.

Since both $L_1(S_1)$ and $L_1(S_2)$ are no less than U, both states are pruned. As a result, the algorithm terminates claiming that the optimal plan is obtained by merging P_{11} and P_{21}.

If the heuristic function L_2 is used, then the following computation is done. First, the H_i functions are computed for the initial state S_0:

$$\begin{aligned} H_1(S_0) &= \{A(P_{11}), A(P_{12})\} \\ &= \{\{a, b, c\}, \{d, e\}\}, \\ H_2(S_0) &= \{A(P_{21}), A(P_{22}), A(P_{23})\} \\ &= \{\{a', f, g\}, \{h, b', i\}, \{j, e'\}\}. \end{aligned}$$

The compution with the heuristic function L_2 involves the following:

$L_2(S_1)$:

$$H_1(S_1) \text{ contains an empty set,}$$

$$H_2(S_1) = \{\{f, g\}, \{h, i\}, \{j, e'\}\},$$

Therefore,

$$\begin{aligned} L_2(S_1) &= \text{cost}(P_{11}) + \min\{15, 20, 40\} \\ &= 55. \end{aligned}$$

$L_2(S_2)$:

$$H_1(S_2) \text{ contains an empty set,}$$

$$H_2(S_2) = \{\{a', f, g\}, \{h, b', i\}, \{j,\}\},$$

Therefore,

$$\begin{aligned} L_2(S_2) &= \text{cost}(P_{12}) + \min\{35, 35, 30\} \\ &= 75. \end{aligned}$$

Since both $L_2(S_1)$ and $L_2(S_2)$ are no less than U, both states are pruned. As a result, the algorithm terminates claiming that the optimal plan is obtained by merging P_{11} and P_{21}.

Notice that with both heuristic functions L_1 and L_2, the number of states generated is 3, although the total number of states in the search space is 9, and the number of states along the path from the initial state to the goal state is 5.

8.3 Experimental Results

In the worst case, Algorithm 8.2 takes exponential time. Since the multiple-goal plan optimization problem is NP-hard, this is not surprising. What would be more interesting is how well Algorithm 8.2 does on the average. However, the structure of the multiple-goal plan optimization problem is complicated enough that it is not clear how to characterize what an "average case" should be; and there is evidence that the "average case" will be different for each application area. Therefore, we have restricted ourselves to doing empirical studies of Algorithm 8.2's performance on a class of problems that seemed to us to be "reasonable."

Experiments with the algorithm have been conducted for planning in the automated manufacturing domain. The problem to be solved involved planning how to drill several holes in a piece of metal stock, as described in example 6.2. For the test, the procedure was to create randomly generated sets of holes with varying machining requirements (such as depth, surface finish, etc.), and to generate plans for each hole individually using EFHA [44], a process planner which can generate one or more alternate plans for creating a machined feature. For each set of holes, Algorithm 8.2 was invoked on the plans for these holes to produce a global plan with the least cost. The total cost of a process plan is the sum of the costs of operations in the plan and the costs for changing tools.

The input plans P_{ij}, $j = 1, 2 \ldots$, for making each hole G_i are ordered in the increasing values of their costs. Once the plans for all the holes are gathered, the upper bound U is computed by combining and merging the locally best plans P_{i1} for each hole, $i = 1, 2, \ldots, g$. As was pointed out before, this computation takes time $O(m^3)$, where m is the total number of actions in the locally best plans. Notice that U is the cost of a leaf node S_U in the search tree, where S_U is the state at the bottom left corner of the search tree shown in Fig. 8.2. Notice also that the computation of S_U does not involve the expansion of any states in the search tree, and therefore, the computation is less expensive than searching

down the left most branch of the search tree in Fig. 8.2. During the execution of the branch-and-bound algorithm, any state with a cost greater than or equal to U is cut off. If the plans P_{i1}, $i = 1, 2, \ldots, g$, can be merged into a least cost global plan, then it is possible that all branches can be quickly cut off, even before the bottom of the tree is reached. In such a case, it is possible that the total computation is less expensive than searching down the optimal branch of the tree.

The experimental result is given in table 8.2. It was done with a specific cost for performing one tool change operation, and each entry in the table represents an average result over 10 randomly generated problems. In the table, the attribute "Holes" is the number of machined holes to be planned for. Attribute "With L2" is the number of states expanded by the Algorithm 8.2 with heuristic function L2. "Without L2" is the number of states expanded using only the total cost accumulated so far as the guide in best first searches. "Goal-Path" is the number of states along the path from the root of a search tree to the goal state. And finally, "Total" is the total number of states in the whole search tree. Notice that the number of states expanded with L2 can sometimes be less than the number of states along the path from the root to the goal node. As explained above, this is because the upper bound U is computed by merging and combining the locally optimal plans. When the set of locally optimal plans is also the globally optimal, most of the paths can be very quickly cut off by U.

The experimental results are also plotted in Fig. 8.4. As shown in the figure, the number of possible states in the search space grew exponentially as a function of the number of holes in the set, but the number of states searched by Algorithm 8.2 with the heuristic function L2 grew only linearly. Intuitively, this means that the portion of the search space explored by our search algorithm is a narrow region around the optimal path from the root of the search tree to the goal state.

unit-tool-change-cost= 50				
HOLES	WITH L2	WITHOUT L2	GOAL-PATH	TOTAL
5	2	16	12	149
10	16	53	24	8381
15	45	70	38	1924861
20	58	135	52	268569213
25	67	312	65	25479050199
30	90	1773	80	8196661999742
35	98	2164	91	545251761730719

Table 8.2: Number of states expanded when *unit-tool-change-cost*=50.

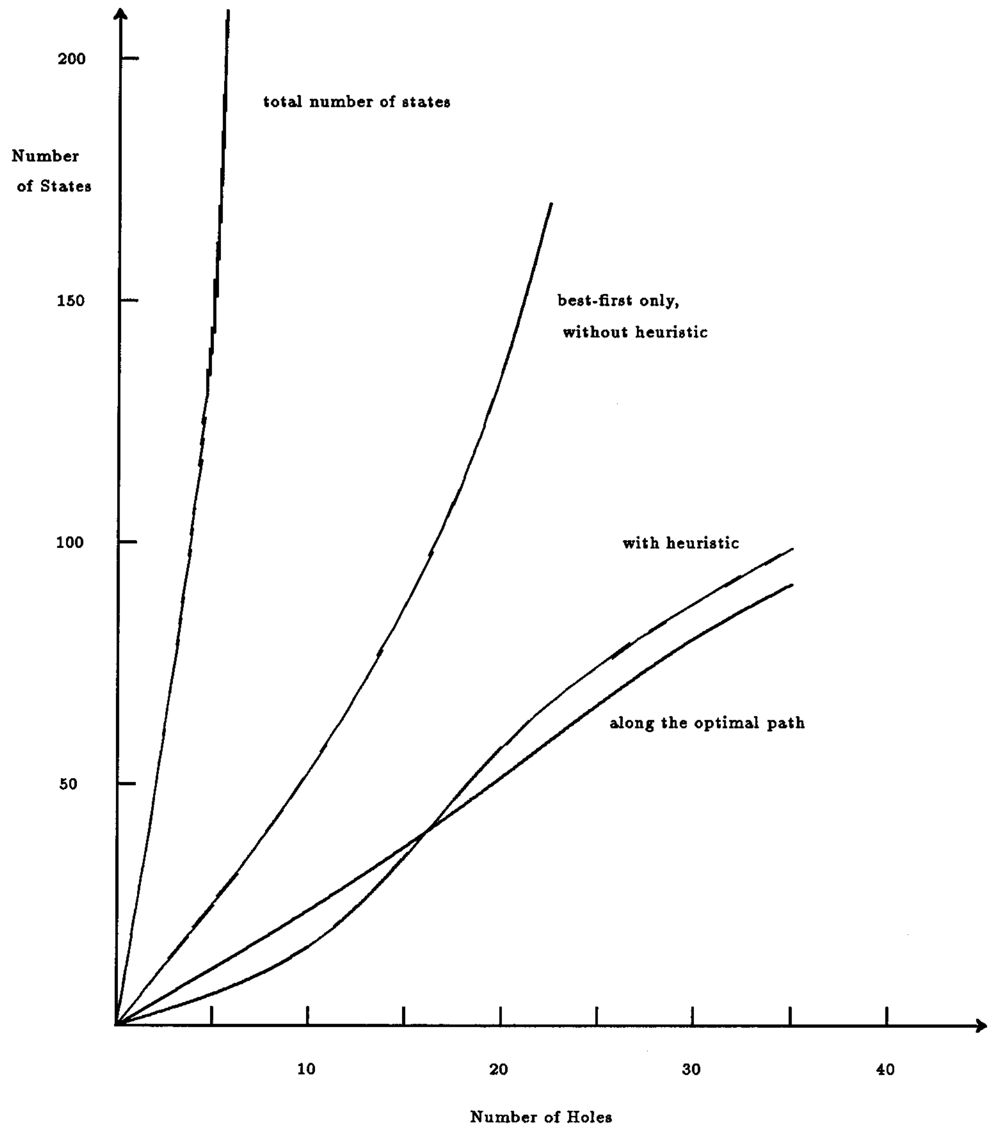


Figure 8.4: Experimental results of merging process plans for making machined holes, when *unit-tool-change-cost*=50.

Chapter 9

Preprocessing Search Spaces for Single Goal Planning

9.1 Introduction

In Chapters 7 and 8, we have discussed how plans for multiple goals can be constructed when the interactions among the goals are limited. Our approach has been to develop plans for each of the individual goals separately, and then combine the plans and resolve the interactions. For this approach to be applicable, efficient planning systems must exist for solving each goal individually. In many cases, each individual goal can be solved via a straightforward heuristic search method. In this chapter, we discuss a method to preprocess the search spaces so that planning for each goal can be done faster than using the conventional methods.

Most heuristic search procedures (for example, A*, SSS*, B*, AO*, and alpha-beta) have been shown to be special cases of best-first Branch-and-Bound search [22, 30]. These procedures search several solution paths at the same time, expanding some paths more

quickly than others depending on which paths look most promising. One source of computational overhead during such search is the time spent keeping track of the alternate partial solutions that the procedure is examining. Typically, the search algorithm stores these partial solutions on a list called the *agenda*, *open list*, or *active list*, in order of their estimated cost. Any time a new partial solution is generated, its estimated cost must be compared with the estimated costs of the other partial solutions already on the list, in order to find the appropriate place to put it on the list. Thus, significant computational overhead is required just to maintain the active list.

Certain problem domains have special properties which allow us to eliminate this overhead. In particular, in some problem domains it is possible to do automatic preprocessing or "compiling" of the search space, to extract control knowledge which can be used to do the heuristic search without having to maintain the active list explicitly. In this chapter, we discuss what kind of domains allow us to do such preprocessing, how to do the preprocessing, how to use the information gathered from the preprocessing, and how much time can be saved by doing this preprocessing. We also present an example from a problem domain of particular interest to us: generative process planning for the manufacture of machined parts.

9.2 Problem Characteristics

Preprocessing of the search space is possible whenever the following conditions are satisfied:

1. With the exception of feasibility/infeasibility of nodes, the search space has the same shape regardless of the particular problem instance being considered. Thus, if P and P' are two problem instances, then there is a one-to-one mapping between the nodes of their search spaces S and S', such that if some feasible node a in S has n children b_i, $i = 1 \ldots n$, then its corresponding node a' in S' will either have n children b'_i, for $i = 1 \ldots n$ such that b_i corresponds to b'_i, or be infeasible.

2. Corresponding nodes need not have the same cost—but if two nodes a and b in S_1 have $\text{cost}(a) > \text{cost}(b)$, then for the corresponding nodes a' and b' in S', we must have $\text{cost}(a') > \text{cost}(b')$.

Nearly all heuristic search problems satisfy the first condition above, but fewer satisfy the second one. However, there are still many problems which do satisfy it. One example is generative process planning for the manufacture of machined parts. In the generative process planning system SIPS[31], a machinable part is considered to be a collection of machinable features. For each feature, SIPS finds an optimal plan for that feature via a best-first Branch-and-Bound search in its knowledge base. SIPS's knowledge base consists of information on a large number of different machining processes, organized in a taxonomic hierarchy. For two different machinable features, different machining processes may have different costs–but if process a costs more than process b on feature f, it generally also costs more than process b on feature f' as well.

Robotic route planning[23] is another problem domain which also satisfies the second condition. In this domain, a route map corresponds to a graph, in which a node represents a region and an arc represents connectivity between two regions. The cost of a node can be the time needed to pass through the region it represents, and the feasibility of a node indicates whether or not the region is passable. In general, although the passability of a region may change from one problem instance to another, the relative costs of the regions stay the same. In such cases, preprocessing is possible.

9.3 Threaded Decision Graphs

Suppose that for a given problem instance the search space is as shown in Fig. 9.1. The goal is to search for the cheapest leaf node for which all the nodes along the path from the root are feasible. Suppose the problem solver is checking the conditions associated with node c. If c succeeds, thent the next node to check will be f, since c must be the current minimum costly node to be checked, and f's cost is the same as c's. If c fails, the next node to check in the search space must be j. This is because at the time c is being checked, nodes a, b, d, e, h and k must all have been checked, since they all have costs less than that of c's. Control information like this is independent of the particular problem instances, and can be gathered before the problem solving process starts. Therefore, we can assign a *success thread* from node c to node f, and a *failure thread* from c to j. If node c is reached, and if all the conditions are satisfied, then the success thread of c can

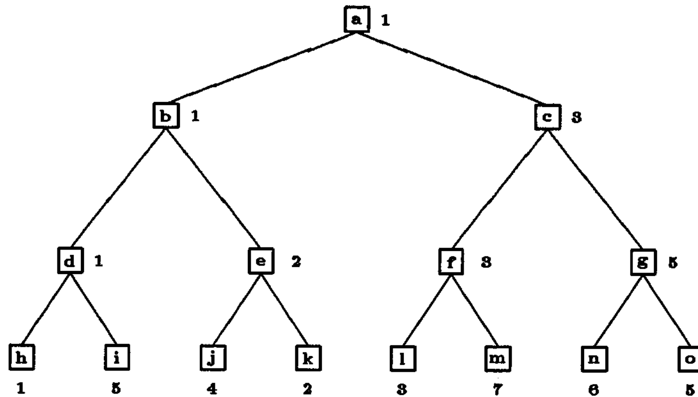


Figure 9.1: An example search space. The numbers associated with the nodes are their cost values.

be followed to get to the next node to be checked. On the other hand, if the conditions are not satisfied, then the failure thread of *c* should be followed to get to the next node to be checked. Fig. 9.2 shows such a structure. In this figure, the solid arrow is *c*'s success thread, and the dotted arrow is its failure thread.

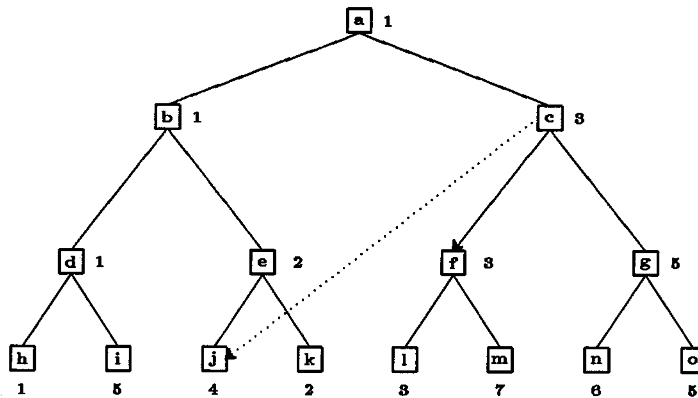


Figure 9.2: The search space in Fig. 9.1 with threads inserted at node c.

If all the success and failure links are built for a state space, then it is no longer necessary to maintain an active list for storing control information. Before problem solving starts, the information can be used to construct a data structure that contains one or more

occurrences of each action, and for each action occurrence there is a success and a failure thread. We will call such a data structure a *threaded decision graph* of the original search space. Problem solving can then be completely guided by its threaded decision graph. Fig. 9.3 shows the threaded decision graph for the search space in Fig. 9.1. In this figure, a solid arrow represents a success thread, and a dotted arrow a failure thread.

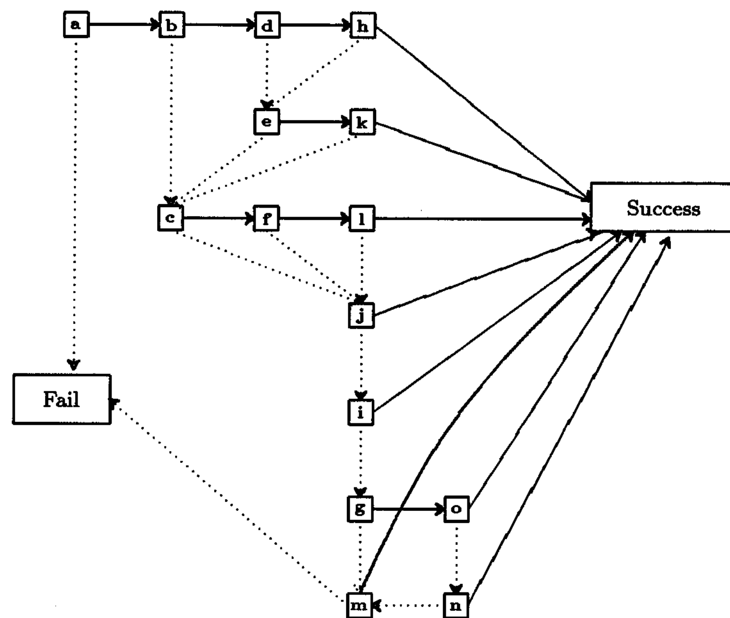


Figure 9.3: The threaded decision graph for the search space in Fig. 9.1.

Two special nodes in Figure 9.3 are worth noting. The node marked "success" is called a *success node*. If this node is reached through the success threads, then the search process terminates with success. The path from the root node to success in the threaded decision graph contains the solution path which should be returned. The node marked "fail" is called a *failure node*, which marks the termination of the search process without success.

In the following, we first discuss in detail how problem solving is done with the help of threaded decision graphs. We then consider how such a data structure could be auto-

matically constructed.

9.4 Search with the Threaded Decision Graphs

Assume that a threaded decision graph has been constructed for a search space. Associated with each node, the following information is attached.

1. a success and a failure thread,
2. feasibility conditions, and
3. a list of the node's ancestors in the search tree.

A request to solving the problem would correspond to finding out the cheapest path in the tree. For the given threaded decision graph, this is done by checking the conditions associated with each node, starting from the root node. The success thread of the current node should be followed if the node is tested to be feasible, and the failure thread should be followed if it is infeasible. Each node is also marked as "success" or "failure" depending on the result of its feasibility test.

If, during the traversal process, a success node is reached via a success thread, then the search ends with success. If a failure node is reached through failure thread, then no actions satisfying the criteria exist. In this case, search is terminated with failure.

After tracing down the failure thread of some node, it is possible to reach some other node whose ancestor in the original search tree failed the feasibility test. For example, suppose node b in figure 3 is infeasible and node f is being checked. If f is infeasible then the next node to be checked should be j. Since b is an ancestor of j which failed, no feasibility test need be done at j, and the failure thread of it should automatically be followed. In general, whenever a node n is reached via some failure thread, it should always be checked to see if n has any ancestor m which failed the feasibility test. If so, the failure thread of node n should be followed, without conducting its feasibility test.

To check if a given node n has any infeasible ancestor is straightforward: we can retrieve the traversal information associated with the parent node of the given node in the original search space. The node n has an infeasible ancestor if its parent node is unvisited or marked as infeasible.

9.5 The Construction of Threaded Decision Graphs

In this section, we consider the problem of how the threaded decision graphs are constructed. For simplicity, it is assumed that the search space is not very large. Later we will discuss cases where the search space is too large to be preprocessed.

With the above restrictions, the following algorithm generates the threaded decision graph. This algorithm is a modified version of best-first search.

```
procedure Construct
  begin
      A := { a0 }        {A is the active list, and a0 is the root of the search space}
      while A is not empty do
      begin
          n := pop(A).
          if A is empty then Failure-thread(n):=Failure-node
          else Failure-thread(n):= head(A);
          if n is a leaf node, then Success-thread(n):=Success-node
          else begin
                  A:=insert(A, Children(n));   {Insert the successors of n into A.}
                  Success-thread(n):=head(A);
          end{else}
      end{while}
end Construct
```

During preprocessing of the search space, each node in the tree appears at the head of the active list once and only once, and the resultant threaded decision graph has the same number of nodes as the original search space. Each node has only two threads. Therefore,

the number of threads of the threaded decision graph is twice the number of nodes as that of the search space. This argument guarantees the absence of cycles of threads.

9.6 Very Large or Infinite Search Spaces

If the search space is very large, the time it takes to preprocess it may be prohibitive. In such cases, and in cases when the search space is infinite, we can preprocess a finite portion of it. This partial threaded decision graph can be created by running the algorithm *Construct* for a finite number of iterations, until one or more goal states appear in the graph.

During the problem solving process, the threaded decision graph can be used in the same manner as described in the previous sections, until a node which has no success and failure threads is reached. At this point, conventional branch-and-bound search can be utilized as follows: An active list is constructed by including all of the successor nodes of the current node and the children of the feasible nodes along the path back to the root node in the state space. This list is sorted according to the costs of its nodes, and the least costly of these is expanded.

The construction of the threaded decision graph may also be done with the following strategy: For each particular problem instance, every time a node is tested we will look for its success or failure threads. If they do not exist, we will use the search procedure to find out which node we should test next. The success or failure threads can then be assigned to this node. In other words, the threaded decision graph is partially built on the job.

9.7 Discussion

By gathering all the possible static control information beforehand, preprocessing of the search space can greatly improve problem solving efficiency. In order to see this claim more quantitatively, consider a simplified example. Assume that the search space is in the shape of a tree with a branching factor of m and depth k (the root of the tree has a depth of 1). To see how much time the problem solver spends on manipulating the list of

incomplete paths using conventional search methods, consider the following two extreme cases.

In the best case, the search procedure expands only one path down the tree. The time for manipulating the active list is $O(km \log m)$.

In the worst case, the search procedure expands the tree in a breadth first manner. The total time spent on manipulating the list in this case is bounded by $m \times \sum_{i=1}^{m^{(k-1)}} \log(i \times m) = O(m^k \log m^k)$.

On the other hand, search with a threaded decision graph is guided completely by the success and failure threads, and no additional computational effort is needed for manipulating the search control information. Therefore, the time saved by using the threaded decision graph to guide the search procedure can be between $O(km \log m)$ and $O(m^k \log m^k)$, where k is the depth of the tree and m is the branching factor.

The idea of preprocessing search spaces has some similarity to that of threading of binary trees for tree traversal. However, the way such threading is done, and the way it is used, are both clearly different from our threaded decision graphs.

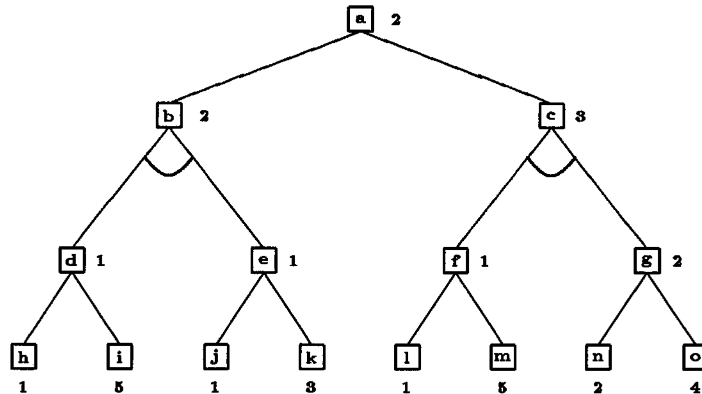


Figure 9.4: An example AND/OR tree in the worst case situation in terms of preprocessing.

As described in the chapter, the technique of preprocessing search space works well on state-spaces that are trees. An extension of the work is to allow the search space to be an AND/OR graph. When AND branches are allowed, the problem can be very complicated depending on how the costs are assigned its nodes. In the worst case, the threaded decision

graph can contain an exponential number of the occurrences of nodes in the original search space. An example of this situation is given in Fig. 9.4. The problem is that the ith and $i+1^{th}$ best paths, for $i = 1, 2, \ldots$, are not next to each other, but on the left and right subtrees respectively. One way to tackle this problem is to *partially* preprocess the search space. For example, we can assign success and failure threads only to the nodes which are in the first k best paths.

Chapter 10

Conclusion

10.1 Summary

Symbolic planning involves extensive reasoning about interactions among different parts of plans. Past planning systems can be classified as domain-independent or domain-dependent according to their method of detecting and handling goal interactions. Domain-independent planners focus on control structures capable of working in all domains, and are for the most part inefficient because of the generality they try to achieve in dealing with interactions. Domain-dependent planners, on the other hand, make use of domain-specific knowledge and have been more efficient than their domain-independent counterparts. However, because of their problem-specific nature they are difficult to apply to other domains.

This thesis presented two approaches to improving the efficiency of planning without losing the generality of planning techniques. In Part I, it was demonstrated that useful information about conflicts between actions can be extracted by preprocessing the planning knowledge. In particular, a set of syntactic restrictions is imposed on the relationship between a non-primitive action and its set of subactions. These restrictions entail a number

of properties for hierarchical planning systems. First, it is possible to determine that a plan is unlinearizable at any level of reduction below, given that it is unlinearizable at the current level. If this downward-unlinearizability property holds for a set of action reduction schemata, hierarchical planners need not reduce an unlinearizable plan further to explore possible solutions at some level or reduction below, and can backtrack from such a dead end point. Moreover, because of the static nature of the restrictions, it is possible to preprocess a given set of reduction schemata to check which schema satisfy the restriction before the planning process starts. Second, it has also been possible to define another property, the ordering-induced independence property, which holds whenever the condition-promotion restriction is satisfied. Like the downward-unlinearizability property, the ordering-induced independence property allows preprocessing of a set of action reduction schemata, and computational savings at planning time.

In Part II, limited-interaction planning was proposed to overcome the shortcomings of both domain-dependent and domain-independent planning methods. This approach is to abstract the kinds of goal and subgoal interactions that occur in some set of related problem domains, and develop planning techniques capable of performing relatively efficiently in all domains in which no other kinds of interactions occur. The limited-interaction planning approach was applied to the multiple-goal planning problems. Four kinds of goal interactions were allowed: action-precedence, identical-action, simultaneous-action and action-merging. It was demonstrated that the multiple-goal plan optimization problem is NP-hard even when only one plan is available for each goal. However, it is possible to impose restrictions upon the action-merging interactions so that polynomial time algorithms can be constructed to solve the problem. When more than one plan exists for each goal, the problem is still NP-hard even with the restrictions. But in this case there is a good heuristic approach for solving the problem.

Chapter 9 presents a method for improving the planning efficiency for solving each individual goal, by gathering all the possible static control information beforehand. This information is stored in a data structure called the *threaded decision graph.* During a search, the control information for manipulating the list of incomplete paths can be obtained from this graph. Since no time is spent on processing the list of incomplete paths, problem solving efficiency is improved. It is interesting to note that the idea of preprocessing underlying Chapter 9 is similar to that of Part I.

Part I and II are related in several aspects. First, the type of hierarchical planning considered in Part I is a special case of planning with limited interactions, in which the only type of interaction allowed is deleted-condition conflict between two actions. Second, the approach to multiple goal plan optimization in Part II is to first plan for each goal separately, and then combine the results into a global plan. Thus, planning for each goal can be done with a hierarchical planner, while the optimization process is done on the same level of planning hierarchy.

In the next section, we discuss limitations of the current research and possible future work.

10.2 Limitations And Future Work

10.2.1 More Expressive Action Representations

In Chapters 4 and 5, several properties of hierarchical planning were considered with a particular kind of action representation, in which the preconditions and effects of an action are sets of literals in first order logic. This restricts the techniques developed to be only useful for a subset of the existing hierarchical planning systems. For example, SIPE [47] allows in its action representation conditions which can be context-dependent, quantified, and disjunctive. In addition, some effects can be deduced from a set of external axioms. With action representations as expressive as these, some of our result does not apply anymore. For example, consider actions which preconditions contain disjunctions of conditions. Then the effects of a group of actions can collectively deny the preconditions of other actions. Thus, the ordering-induced independence property in Chapter 5 may not hold anymore with the given restrictions, since although the effects of the descendents of an action does not directly deny the precondition of another action, a set of actions can. With this action representation, the detection of interactions during planning is a more complicated problem, since at every step of planning, groups of actions have to be tested for possible interactions. We believe that the same idea of extracting information about interaction *a priori* can still be used to help ease this computation burden, in the same way as we did in this thesis. For example, information about which actions are more likely to interact may be obtained through preprocessing, and used to provide a better control

strategy for the detection and handling of interactions.

10.2.2 Choosing Among Alternative Reduction Schemata

Preprocessing can provide a planner with good heuristics for choosing among alternative action reduction schemata. To reduce a non-primitive action, there is usually more than one available schemata, and current hierarchical planning systems selects the alternatives in a depth-first manner. Depending on which reductions are chosen, different conflicts may be introduced, and this may affect the planning efficiency.

One heuristic one can use in choosing a reduction is to minimize the amount of conflicts introduced. Preprocessing can be helpful for predicting which reductions are likely to yield a set of conflicts that is small—and just as importantly, a set of conflicts which are easy to resolve.

For example, consider a plan which consists of two actions a and b, with no constraints on the relative ordering of a and b. Suppose that action a has two possible reductions $R_{11}(a)$ and $R_{12}(a)$, and the action b has also two reductions $R_{21}(b)$ and $R_{22}(b)$. For each reduction $R_{ij}(x)$ of action x, let preconditions($R_{ij}(x)$) be the set of preconditions for the subactions in $R_{ij}(x)$ which are not achieved by any subactions in the reduction. Also let all-effects($R_{ij}(x)$) be the union of the effects for the all subactions, and let protected($R_{ij}(x)$) be the set of conditions protected in $R_{ij}(x)$. Then the number of possible conflicts between $R_{1i}(a)$ and $R_{2j}(b)$ can be estimated by

$$\begin{aligned} C(R_{1i}(a), R_{2j}(b)) = \quad & |\{p \in \text{protected}(R_{1i}(a)) : \neg p \in \text{protected}(R_{2j}(b))\}| + \\ & |\{p \in \text{preconditions}(R_{2j}(b)) : \neg p \in \text{all-effects}(R_{1i}(a))\}| + \\ & |\{p \in \text{preconditions}(R_{1i}(a)) : \neg p \in \text{all-effects}(R_{2j}(b))\}|. \end{aligned}$$

According to the heuristic discussed above, the reductions to be chosen for actions a and b are those $R_{1i}(a)$ and $R_{2j}(b)$ such that $C(R_{1i}(a), R_{2j}(b))$ is minimum for all $i, j = 1, 2$.

One major problem with searching for a good set of reductions is that the search itself also takes time. Let T_1 be the amount of time needed to remove the conflicts presented by the "worst" set of actions (i.e., the set of actions presenting the largest number of conflicts). Let T_2 be the same figure for the best set of actions (i.e., the set of actions presenting the smallest number of conflicts). Also, let T_3 be the time needed to search for

the best set of actions. Then finding the best set of actions is useful if and only if

$$T_1 > T_2 + T_3.$$

If this condition is not satisfied, then we may just want to find any set of actions, rather than looking for the best set. There are two specific problems to solve. First, T_3 will presumably be dependent upon the way action reduction schemata are constructed. Thus, restrictions need be set up for guaranteeing that T_3 be small. Second, in domains where T_3 is prohibitively large, it may be better for the planning system to look for a suboptimal set of reduction schemas, and algorithms for computing the suboptimal solutions need be designed.

10.2.3 Computing Planning Orderings

During hierarchical planning, a plan often has more than one non-primitive action to be reduced. A least-commitment planner delays commitment to both decisions about orderings between the actions and bindings of the variables. Therefore, a plan usually contains a set of variables yet to be bound. The order in which the non-primitive actions are reduced may affect how the variables are bound in a plan, which in turn affects both the efficiency of planning and the quality of the final plan. The order in which the non-primitive actions are reduced can be called *planning order*, which may be quite different from the temporal orderings.

Smith [39] presented a method for determining the planning order for a set of conjunctive goals. His method is based on a cost model for achieving each atomic condition, and is applicable only to non-hierarchical systems. It would be useful to extend his work to hierarchical planning, in which a correct planning order may depend on both the cost model for actions and the way a non-primitive action relates to its subactions. Thus, a good starting point for extending Smith's work is to impose syntactic restrictions on the reduction schemata, in a way similar to what was done in Part I.

10.2.4 Relaxation of Some Restrictions on Multiple Goal Planning

Part II concerns improving the efficiency of multiple goal planning by imposing restrictions on the goal interactions. One question which of interest is whether some of the particular

restrictions are too restrictive. Relaxing certain restrictions imposed on multiple goal planning domain may require the imposition of some other restrictions in order to keep the problem at low complexity. A related question is how to extend this approach to creating plans, rather than just optimizing existing plans.

As an example of the relaxation of restrictions, consider the restriction upon action-merging interactions requiring that the merged action $m(A)$ for a set of actions A does not have any harmful effects. Lifting this restriction would make it necessary to modify our control mechanism correspondingly. For example, if in addition to achieving all the effects of the actions in A, $m(A)$ also achieves $\neg p$, where p is one of the preconditions of another action a'. If $m(A)$ and a' are unordered in the plan, then there is a conflict between these two actions. In this case, it may or may not worth while to merge the actions in A, depending on how easy it is to handle the deleted-condition conflict. In general, it may be necessary to insert new actions to re-achieve some deleted conditions as a result of merging. Thus, it is necessary that certain elements of "traditional" planning techniques have to be added to multiple goal planning systems if some restrictions are relaxed.

The multiple goal planning problem considered in the thesis concerns finding the least cost multiple goal plan for a set of goals. Other types of multiple goal planning problems exist, each of which may have a different optimality criterion. For example, consider the problem of finding a global plan with the minimum number of ordering constraints between the individual plans. Such a problem can be referred to as the *most parallel multi-goal planning problem.* When the goals interact with each other in a limited fashion, it is expected to be more efficient to plan for each goal first, and then combine them into a global plan with the minimum number of ordering constraints. For this approach to be applicable, again one needs to be clear about what kind of goal interactions are allowed. This problem is of practical importance, since maintaining maximal parallelism allows one to make use of as many available agents as possible to save time. Research is under way to solve this problem.

10.2.5 Suboptimal Multiple Goal Plans

The multiple goal planning problem is defined as the problem of finding the least-cost plans. To solve this problem, two different cases have been distinguished, based on whether

there is more than one plan for each goal. In both cases, branch-and-bound algorithms are given that take as input a set of plans and return the least-cost plan. Several implicit assumptions have to hold for this formulation to work. One of the assumptions is that there is no time limit on the optimization algorithm. Therefore, the algorithms can wait until all the the alternate plans are available, and then perform the optimization. There may be some practical situations when such a assumption is invalid. For example, in an environment where decisions have to be made within a certain time limit, one may be satisfied with a good suboptimal plan, but cannot always afford to wait for the computation of the optimal one. In this case, good approximation algorithms are needed. Thus, one of our future work is to design polynomial time algorithms that return good plans on the average.

10.2.6 Preprocessing Search Spaces

Chapter 9 presented an approach to preprocess search spaces to improve the problem solving efficiency. One underlying assumption for this technique is that the relative costs associated with the states stay the same for a class of problems. This is satisfied by a number of problems. For some others, the relative costs of some nodes in different problems may not be the same, but may change with different situations. However, if the cost of nodes change with different situations in some predictable manner, and if the number of such changes is finite, the preprocessing technique can still be made to work. For example, it may be possible to divide the situations into separate classes, each with a different threaded decision graph. Many domains satisfy this property, including process planning in automated manufacturing and robotic path planning.

Appendix A

Action Templates and Reduction Schemata for the Blocks-World Domain

This appendix provides a representation of the planning knowledge in the blocks-world domain[1]. In this domain, there are a table and a number of equally sized blocks. A block can have at most one other block immediately on top of it. A robot can stack blocks on top of each other, or put a block on the table. The table is considered to be always clear. The robot can handle only one block at a time.

The following actions are non-primitive:

1. achieve(On(x, y))

   comment: a goal for achieving On(x, y).

   preconditions={}

   effects={On(x, y)}

[1] This domain representation is adopted from [37].

2. achieve(**Ontable**(x))

   comment: a goal for achieving Ontable(x).

   preconditions={}

   effects={Ontable(x)}

3. achieve(**Cleartop**(x))

   comment: a goal for achieving Cleartop(x).

   preconditions={}

   effects={Cleartop(x)}

4. **makeon-block-1**(x, y)

   comment: put block x on the top of block y.

   preconditions={Block(x), Block(y), Cleartop(x), Cleartop(y)}

   effects={$\neg$Cleartop(y), On(x, y)}

5. **makeon-table-1**(x)

   comment: move x to the table

   preconditions={Block(x), Cleartop(x)}

   effects={Ontable(x)}

6. **makeon-block-2**(x, y, z)

comment: move x from the top of y to the top of z.

preconditions=$\{$Block(x), Block(y), Block(z),

On(x, y), Cleartop(z), Cleartop(x)

$x \neq y, x \neq z, y \neq z\}$

effects=$\{$Cleartop(y), On(x, z), $\neg$Cleartop(z)$\}$

7. **makeon-table-2(x, y)**

   comment: move x from top of y to the table.

   preconditions=$\{$Block(x), Block(y), Cleartop(x), On(x, y)$\}$

   effects=$\{$Ontable(x), Cleartop(y)$\}$

The following actions are primitive:

1. **put-block-on-block(x, y, z)**

   comment: move x from the top of y to the top of z.

   preconditions=$\{$Block(x), Block(y), Block(z),

   On(x, y), Cleartop(z), Cleartop(x)

   $x \neq y, x \neq z, y \neq z\}$

   effects=$\{$Cleartop(y), On(x, z), $\neg$Cleartop(z), $\neg$On(x, y)$\}$

2. **put-block-on-table(x, y)**

   comment: move x from top of y to the table.

   preconditions=$\{$Block(x), Block(y), Cleartop(x), On(x, y)$\}$

   effects=$\{$Ontable(x), Cleartop(y), $\neg$On(x, y)$\}$

Below are action reduction schemata:

1. $R_1(\text{achieve}(\text{On}(x, y)))$:

   actions: $\{\text{achieve}(\text{Cleartop}(x)), \text{achieve}(\text{Cleartop}(y)), \text{makeon-block-1}(x, y)\}$

   orderings: $\{\text{achieve}(\text{Cleartop}(x)) \prec \text{makeon-block-1}(x, y)$

   $\text{achieve}(\text{Cleartop}(y)) \prec \text{makeon-block-1}(x, y)\}$

   protection intervals: $\{\langle \text{Cleartop}(x), \text{achieve}(\text{Cleartop}(x)), \text{makeon-block-1}(x, y)\rangle$

   $\langle \text{Cleartop}(y), \text{achieve}(\text{Cleartop}(y)), \text{makeon-block-1}(x, y)\rangle\}$

2. $R_2(\text{achieve}(\mathbf{Cleartop}(x)))$:

   actions: $\{\text{achieve}(\text{Cleartop}(y)), \text{makeon-block-2}(y, x, z)\}$

   orderings: $\{\text{achieve}(\text{Cleartop}(y)) \prec \text{makeon-block-2}(y, x, z)\}$

   protection intervals: $\{\langle \text{Cleartop}(y), \text{achieve}(\text{Cleartop}(y)), \text{makeon-block-2}(y, x, z)\rangle\}$

3. $R_3(\text{achieve}(\mathbf{Cleartop}(x)))$:

   actions: $\{\text{achieve}(\text{Cleartop}(y)), \text{makeon-table-2}(y, x)\}$

   orderings: $\{\text{achieve}(\text{Cleartop}(y)) \prec \text{makeon-table-2}(y, x)\}$

   protection intervals: $\{\langle \text{Cleartop}(y), \text{achieve}(\text{Cleartop}(y)), \text{makeon-table-2}(y, x)\rangle\}$

4. $R_4(\text{achieve}(\mathbf{Ontable}(x)))$:

   actions: $\{\text{achieve}(\text{Cleartop}(x)), \text{makeon-table-1}(x)\}$

   orderings: $\{\text{achieve}(\text{Cleartop}(x)) \prec \text{makeon-table-1}(x)\}$

protection intervals: $\{\langle \text{Cleartop}(x), \text{achieve}(\text{Cleartop}(x)), \text{makeon-table-1}(x) \rangle\}$

5. R_5(**makeon-block-1**(x, y)) :

   actions: {put-block-on-block(x, z, y)}

   orderings: {}

   protection intervals: {}

6. R_6(**makeon-block-2**(x, y)) :

   actions: {put-block-on-block(x, z, y)}

   orderings: {}

   protection intervals: {}

7. R_7(**makeon-table-1**(x)) :

   actions: {put-block-on-table(x, y)}

   orderings: {}

   protection intervals: {}

8. R_8(**makeon-table-2**(x, y)) :

   actions: {put-block-on-table(x, y)}

   orderings: {}

   protection intervals: {}

In addition to the above reduction schemata, every non-primitive action of the form "achieve(C(x))" can also be reduced to "no-op", which is a primitive action requiring no real action. no-op corresponds to a phantom goal in NONLIN [42] or SIPE [47], and can be considered as a primitive action whose precondition and effect are both the condition C(x) to be achieved.

Appendix B

NP-Completeness of the Multiple-Goal Plan Existence Problem

The purpose of this appendix is to show that if there may be more than one plan for each goal, then the multiple-goal plan existence problem is NP-complete. To do this, we show that NP-completeness occurs in the special case where the only kind of goal interaction that occurs is the identical-action interaction.

It is easy to see that the problem is in NP, so the proof will be complete if the problem is shown to be NP-hard. We do this by reducing the CNF-satisfiability problem to it.

Given a set U of variables and a collection C of clauses over U, the CNF-satisfiability problem asks whether there is an assignment of truth values to the variables in U which satisfies every clause in C. To reduce this problem to the multiple-goal plan existence problem, we associate a goal G_i with each clause C_i of C. G is the conjunct of the individual goals G_i. For each literal $l_{ij} \in C_i$, we create a plan (a_{ij}, b_{ij}) for the goal G_i. If $l_{ij} = l_{kl}$, then we specify that a_{ij} and a_{kl} must be identical, and b_{ij} and b_{kl} must also be identical. If $l_{ij} = \neg l_{kl}$, then we specify that a_{ij} and b_{kl} must be identical, and b_{ij} and a_{kl} must also be identical.

It is easy to see that that this reduction can be computed in polynomial time. It remains to be shown that (1) if C is satisfiable, then there is a consistent global plan for G; and (2) if there is a consistent global plan for G, then C is satisfiable. These two statements are proved below.

1. Suppose there is an assignment of truth values to the variables in U which satisfies C. Then we construct a set S of plans, one for each goal G_i. For each i, the clause C_i in C contains some literal l_i^* in C_i whose value is TRUE; we let S contain the corresponding plan (a_{ij}, b_{ij}). Suppose that the plans in S cannot be combined into a consistent global plan. Then there are two plans $p_i = (a_{ij}, b_{ij})$ and $p_k = (a_{kl}, b_{kl})$ such that a_{ij} and b_{kl} are constrained to be identical, and b_{ij} and a_{kl} are constrained to be identical. But this means that $l_i^* = \neg l_k^*$, violating our requirement that both l_i^* and l_k^* have the value TRUE. Thus, the plans in S can be combined into a consistent global plan.

2. Conversely, suppose there is a set of plans S which can be combined into a consistent global plan. Then we assign truth values to the variables in U as follows: for each variable $v \in U$, if its corresponding plan is in S, then assign it the value TRUE; otherwise, assign it the value FALSE. Since S can be combined into a consistent global plan, this means that no variable can receive both the values TRUE and FALSE. Furthermore, since S must contain at least one plan for each goal G_i, at least one literal in each clause will receive the value TRUE. Thus, this assignment of truth values satisfies C.

Appendix C

Branch-and-Bound Search

Problem solving involves search. Many different kinds of heuristic search methods have been developed, and many of them (for example, A*, SSS*, B*, AO* and alpha-beta) have been shown be be special cases of best-first Branch-and-Bound search ([22, 30]). This appendix describes an abstract formulation of the branch-and-bound search procedures based on [30].

A general branch-and-bound search problem can be stated as follows:

> For an arbitrary discrete set X ordered by a real valued merit function f whose domain is X, find some $x^* \in X$ such that for all $x \in X$, $f(x^*) \leq f(x)$.

Branch-and-bound procedures decompose the original set X into sets of decreasing size. The decomposition of each generated set S is continued until either of the following happens.

1. Tests reveal that S is a singleton of the set X. In that case, $f(S)$ can be obtained and compared with the currently best member's cost.

2. S is proved not to contain an optimum element. In this case, S is removed from further consideration. In other words, S is pruned.

To give an abstract formulation of branch-and-bound procedures, it is necessary to defined the following functions.

Let A be a subset of the power set of X, i.e., $A \subseteq 2^X$. $\cup(A)$ is defined as

$$\cup\{X_i \mid X_i \in A\}.$$

Let X_i be an element of A. Then $f^*(X_i)$ is defined as the minimum of the costs of the elements in X_i.

A *branching function*, BRANCH(A), is defined as follows:

1. If $X_i \in$ BRANCH(A), then $X_i \subseteq X_j$ for some $X_j \in A$.

2. $\cup$(BRANCH(A)) $= \cup(A)$.

A *dominance relation* D is the binary relation between elements of A. Specifically, let X_i and X_j be two elements of A. Then $X_i D X_j$ if and only if $f^*(X_i) \leq f^*(X_j)$. Thus, if $X_i D X_j$, then X_j should be pruned.

The pruning function PRUNEcan be defined according to the dominance relation D. Define A^D as

$$A^D = \{X_j \in A \mid \exists X_i \in A - A^D \text{ such that } X_i D X_j.\}$$

Then PRUNE(A) $= A - A^D$.

With the above functions defined, an abstract formulation of the branch-and-bound procedure is given below.

Procedure BB

begin

$A := \{X\}$; {Initialize A}

while $|\cup(A)| \neq 1$ **do begin** {loop until A contains only one element}

$A :=$ BRANCH(A);

$A :=$ PRUNE(A);

end

end

An important property of this procedure is: If Procedure **BB** terminates, A contains only an optimal element of X. However, the termination itself is not guranteed yet.

A lower bound function lb can be defined on elements of A as follows. Let X_i be an element of A, then

1. For all $x \in X_i$, $\text{lb}(X_i) \leq f(x)$.
2. for all $x \in X_i$, $\text{lb}(\{x\}) = f(x)$.

A branch-and-bound procedure is *best-first* if the function BRANCHis defined as follows.

$$\text{BRANCH}(A) = (A - \{X_i\}) \cup \text{SPLIT}(\{X_i\}),$$

where X_i is an element of A such that $\forall X_j \in A$, $\text{lb}(X_i) \leq \text{lb}(X_j)$. The function SPLIT is any function satisfying the properties of a branching function.

An important property of a best-first branch-and-bound procedure is that whenever a singleton is selected, Procedure **BB** terminates. In other words, a best-first branch-and-bound procedure is *admissible.*

For the branch-and-bound search algorithms presented in this thesis, a special implementation of the Procedure **BB** are used. This procedure, Procedure **BB′**, performs a best-first branch-and-bound search, and can be considered as a variation of the A* algorithm.

Procedure BB′

begin

$A := (I)$;

{*A is the branch-and-bound active list,*

and I is the initial state.}

$U :=$ upper bound;

loop

$S :=$ pop(A) {*remove the first element of the list*}

if S is a goal state **then return** S

if $\text{lb}(S) \leq U$ **then begin**

B := the successors of S, in order of least L-value first

A := the result of merging the list A with the list B

{*Here two lists are merged into a sorted list.*

This is to maintain A in order of least L-value first.}

end

repeat

end

In Procedure **BB′**, a state S corresponds to an element X_i of A in Procedure **BB**. The successors of S corresponds to the output of the function SPLIT considered before. A goal state is one which contains the optimal element of A. The upper bound function U in Procedure **BB′** should satisfy the condition that if x^* is a solution, then $f(x^*) \leq U$. This value is used for pruning: any state S with $\mathrm{lb}(S) \geq U$ should be removed from further consideration.

Bibliography

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Data Structures and Algorithms,* Addison-Wesley Publishing Co., Reading, Mass., 1983.

[2] K.R. Baker, *Introduction to Sequencing And Scheduling,* Jone Wiley and Sons, 1974.

[3] T.C. Baker, J.R. Greenwood "Star: an environment for development and execution of knowledge-based planning applications" *Proceedings DARPA Knowledge-based Planning Workshop*, Dec. 1987.

[4] Berlin, M., Bogdanowicz, J. and Diamond, W. "Planning and control aspects of the scorpius vision system architecture" *Proceedings DARPA Knowledge-based Planning Workshop*, Dec. 1987.

[5] A. Brown and Gaucus, D. "Propsective Situation Assessment" *Proceedings DARPA Knowledge-based Planning Workshop*, Dec. 1987.

[6] T. C. Chang and R. A. Wysk, *An Introduction to Automated Process Planning Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[7] D. Chapman, "Planning for Conjunctive Goals," *Artificial Intelligence* (32), 1987, 333-377.

[8] E. Charniak and D. McDermott, *Introduction to Artificial Intelligence*, Chapter 9, Addison-Wesley Publishing Co., 1984.

[9] M. R. Cutkoski and J. M. Tenenbaum, "CAD/CAM Integration Through Concurrent Process and Product Design," *Proc. Symposium on Integrated and Intelligent Manufacturing at ASME Winter Annual Meeting*, 1987, pp. 1-10.

[10] T. Dean and M. Boddy, "Reasoning about Partially Ordered Events," *Artificial Intelligence*, (36), pages 375-399, 1988.

[11] R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence* (2:3/4), 1971, 189-208.

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, Bell Laboratories, Murray Hill, New Jersey, 1979.

[13] Garvey, T. and Wesley, L. "Knowledge-based Helicopter Route Planning" *Proceedings DARPA Knowledge-based Planning Workshop*, Dec. 1987.

[14] D.P. Glasson, and J.L. Pomarede "Navigation Sensor Planning for Future Tactical Fighter Missions" *Proceedings DARPA Knowledge-based Planning Workshop*, Dec. 1987.

[15] C. Hayes, "Using Goal Interactions to Guide Planning," *Proc. AAAI-87*, 1987, 224-228.

[16] L. R. Harris, "The Heuristic Search Under Conditions of Error," *Artificial Intelligence* (5), 1974, 217-234.

[17] James A. Hendler, "Integrating Marker-Passing and Problem-Solving: A Spreading Activation Approach to Improved Choice in Planning," Ph.D. Thesis, Brown University, TR-86-01.

[18] E. Horowitz and S. Sahni, "Fundamentals of Computer Algorithms," Computer Science Press, Potomac, MD, 1978.

[19] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Weseley, Reading, Mass., 1968.

[20] R. E. Korf "Planning as Search: A Quantitative Approach," *Artificial Intelligence* (33), 1987, 65-88.

[21] R. E. Korf, "Real-Time Path Planning" *Proceedings DARPA Knowledge-based Planning Workshop*, Dec. 1987.

[22] V. Kumar and L. Kanal, "A General Branch and Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures," *Artificial Intelligence* (21), 1983, 179–198.

[23] T. A. Linden and J. Glicksman, "Contingency Planning for an Autonomous Land Vehicle," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1047–1054, Milan, August, 1987.

[24] T. A. Linden and S. Owre, "Transformational Synthesis Applied to ALV Mission Planning" *Proceedings DARPA Knowledge-based Planning Workshop*, Dec. 1987.

[25] V. Lifschitz, "On the Semantics of STRIPS," *Proceedings of the Workshop on Reasoning About Actions and Plans*, Oregon 1986.

[26] M. Luria, "Goal Conflict Concerns," *Proc. IJCAI*, 1987, 1025-1031.

[27] D. Maier, "The Complexity of Some Problems on Subsequences and Supersequences," *J. ACM* (25), 1978, 322-336.

[28] M. S. Fox, "Constraint-Directed Search: A Case Study of Job-Shop Scheduling," Ph.D Thesis, CMU-RI-TR-83-22, 1983.

[29] D. McDermott *Flexibility and Efficiency in a Computer Program for Designing Circuits*, AI Laboratory, Massachusetts Institute of Technology, Technical Report AI-TR-402, 1977.

[30] D.S. Nau, V. Kumar and L. Kanal, "General Branch and Bound, and Its Relation to A* and AO*," *Artificial Intelligence*, (23), 1984, 29-58.

[31] D. S. Nau, "Automated Process Planning Using Hierarchical Abstraction," Award winner, Texas Instruments 1987 Call for Papers on Industrial Automation, *Texas Instruments Technical Journal*, Winter 1987, 39-46.

[32] D. S. Nau, Q. Yang and J. A. Hendler, "Planning for Multiple Goals with Limited Interactions," Fifth IEEE Conference on Artificial Intelligence Applications, Miami, Florida, March 1989,

[33] D. S. Nau, N. Ide, R. Karinthi, G. Vanecek Jr. and Q. Yang, "Solid Modeling and Geometric Reasoning for Design and Process Planning," AAAI 88 Workshop on Production Planning and Scheduling, St. Paul, August, 1988.

[34] D. S. Nau, V. Kumar, and L. N. Kanal, paper in preparation.

[35] N. Nilsson, *Principles of Artificial Intelligence*, Chapters 7 and 8, Tioga Publishing Co., 1980.

[36] F. P. Preparata and R. T. Yeh, *Introduction to Discrete Structures*, Addison-Wesley, Reading, Mass., 1973.

[37] Subbarao Kambhampati, Ph.D Thesis, University of Maryland, in preparation.

[38] E. D. Sacerdoti, "A Structure of Plans and Behavior," *American Elsevier, New York,* 1977.

[39] D.E. Smith, "A Decision-Theoretic Approch to the Control of Planning Search," Tech. Report No. LOGIC-87-11, Stanford Logic Group, Department of Computer Science, Stanford University.

[40] M. Stefik, "Planning with Constraints (MOLGEN: Part 1)," *Artificial Intelligence*, (16), 1981, 111-140.

[41] G. Sussman, "A Computer Model of Skill Acquisition," *American Elsevier*, New York, 1982.

[42] A. Tate, "Generating Project Networks," *Proc. IJCAI*, 1977, 888-893.

[43] J. Tenenberg, "Abstraction in Planning," Tech. Report TR250, Computer Science Department, University of Rochester, Rochester, NY 14627, May 1988.

[44] S. Thompson, "Environment for Hierarchical Abstraction: A User Guide," Tech. Report, Computer Science Department, University of Maryland, College Park, 1989.

[45] S. A. Vere, "Planning in Time: Windows and Durations for Activities and Goals," *IEEE Transactions on Pattern Analysis and Machine Intelligence* (PAMI-5:3), 1983, 246-247.

[46] R. Wilensky, *Planning and Understanding*, Addison-Wesley: Reading, Massachusetts, 1983.

[47] D.E. Wilkins, "Domain-independent Planning: Representation and Plan Generation," *Artificial Intelligence*, (22), 1984.

[48] D.E. Wilkins, *Practical Planning:Extending the Classical AI Planning Paradigm*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1988.

[49] Q. Yang, D. Nau, and J. Hendler, "Optimizing Multiple Goal Plans with Limited Interaction," Tech. Report UMIACS-TR-88-49, July, 1988.

[50] Q. Yang, D. S. Nau and J. Hendler, "An Approach to Multi-Goal Planning with Limited Interactions," AAAI 1989 Spring Symposium Series, Stanford University, Stanford, CA., March 1989.

[51] Q. Yang and D. S. Nau, "Preprocessing Search Spaces for Branch and Bound Search," to appear at the Eleventh International Joint Conference on Artificial Intelligence, Detroit, Michigan, August 1989. Also Tech. Report No. UMIACS-TR-88-71, University of Maryland, College Park, October 1988.