Justified Plans and Ordered Hierarchies

by

Eugene Fink

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1993

©Eugene Fink 1993

Acknowledgement

I gratefully acknowledge the guidance and encouragement I received from my supervisor, Professor Qiang Yang. His kindness and understanding will always be remembered.

I would like to thank the readers, Professor Derick Wood and Professor Faheim Bacchus for their careful review of the materials contained in this thesis.

I owe thanks to Craig Knoblock and Josh Tenenberg for our discussions about abstraction planning.

To Margie Roxborough
and John Neville

Abstract

The use of abstraction in problem solving is an effective approach to reducing search, but finding good abstractions is a difficult problem. The first attempt to automatically generate a hierarchy of abstraction spaces was made by Sacerdoti in 1974. In 1990 Knoblock built the system ALPINE, which completely automates the formation of a hierarchy by abstracting preconditions of operators. To formalize his method, Knoblock introduced the notion of *ordered* abstraction hierarchies, in attempt to capture the intuition behind "good" hierarchies.

In this thesis we continue the work started by Knoblock. We present further formalization of several important notions of abstract planning and describe methods to increase the number of abstraction levels without violating the ordered property of a hierarchy.

We start by defining the *justification* of a non-linear plan. Justification captures the intuition behind "good" plans, which do not contain useless actions. We introduce several kinds of justification, and describe algorithms that find different justifications of a given plan by removing useless operators. We prove that the task to find the "best possible" justification is NP-complete.

The notion of justified plans leads us to define several kinds of *semi-ordered* abstraction hierarchies, which preserve the "good" properties of Knoblock's ordered hierarchies, but may have more abstraction levels.

Finally, we present an algorithm for automatically abstracting not only preconditions but also effects of operators. This algorithm generates hierarchies with more levels of abstraction than ALPINE, and may increase the efficiency of planning in many problem domains. The algorithm may generate both problem-independent and problem-specific hierarchies.

Contents

Acknowledgement **2**

1 Introduction **1**
1.1 Informal overview 1
1.2 Outline of the thesis 2

2 Definitions, notation, and basic algorithms **4**
2.1 A formal description of a planning domain 4
2.2 Example of a Planning Domain (Tower of Hanoi) 11
2.3 Literal-Representation vs. Variable-Representation 13
2.4 Domain rules 16
2.5 Criticalities 20
2.6 Data structures and basic algorithms 22

3 Justified plans **29**
3.1 Backward justification 30
3.2 Well-justification 34
3.3 Perfect justification 37
3.4 Greedy justification 42
3.5 A spectrum of justified plans 44

4 Ordered and Semi-ordered Abstraction Hierarchies **46**
4.1 Previous work 47
4.2 Backward semi-ordered hierarchies 48
4.2.1 Definition and properties of semi-ordered hierarchies 48
4.2.2 Necessary and sufficient conditions of the semi-ordered property . . . 51
4.2.3 Forbidding and Non-Forbidding Tests 55
4.2.4 Example 58
4.3 Well-, Greedily, and Perfectly Semi-ordered Hierarchies 59
4.3.1 Learning Technique 63
4.4 Conclusion 66

5 Automatically abstracting effects of operators **67**
5.1 A Motivating Example 67
5.2 Ordered Hierarchies with Primary Effects 68

5.3 Automatically Finding Primary Effects 72
5.4 Advantages and Limitations of using Primary Effects 79
5.5 A Robot-Domain Example 86

6 Goal-specific hierarchies **89**
6.1 Goal-specific version of a problem domain 89
6.2 Using a goal-specific domain in planning 92
6.3 Example of a goal-specific domain 94

7 Conclusion **98**
7.1 Summary 98
7.2 Future work 99

A Summary of Notation **101**

B List of algorithms **103**

Chapter 1

Introduction

1.1 Informal overview

Classical Artificial Intelligence Planning is concerned with the problem of finding ways to achieve a desirable situation, called a *goal state*, starting from some initial state of the world. We are able to perform some atomic actions, called *operators*, and wish to find a sequence of operators that leads to a goal state. For example, suppose our goal is to boil a cup of water. The following simple plan solves the goal:

1. Fill a cup with water.
2. Put the cup into a microwave.
3. Turn the microwave on.

The water-boiling plan is quite simple, but real-life planning problems are often much more complicated. Most of them are unsolvable.

A lot of methods for increasing the efficiency of planning were found in the last twenty years. One of the main methods is to use a *hierarchy of abstraction spaces*. This means that first we find some "outline" of a plan, which contains the most important steps of the plan and omits details. Then we refine the plan by inserting details. If it is still incomplete, we insert more details. The process of refining continues until a correct plan is found.

For example, *step 3* of the plan above implies that we must open and close the door of the microwave. While refining the plan, we must state these actions explicitly. Thus, the refined version of *step 3* looks as follows

3a. Open the door of the microwave.
3b. Put the cup into the microwave.
3c. Close the door of the microwave.

An *abstraction hierarchy* defines the "importance" of results of operators. In our case the result "the cup is in the microwave" is more important than the results "the door is open" and "the door is closed". The results of operators are expressed as sets of *literals*. We say that "the door is open" and "the door is closed" are *concrete-level* literals, and we ignore them while building an outline of a plan at the *abstract level* of a hierarchy.

Intuitively, an abstraction hierarchy increases the efficiency of planning by dividing an initial problem into smaller subproblems. Since planning process usually takes exponential running time, such a division may lead to an exponential increase of efficiency [Knoblock, 1991b]. However, the choice of a "good" hierarchy that increases an efficiency of planning is a difficult problem.

The technique of ignoring some literals while planning at the abstract level was first used by Newell and H. Simon in their GPS planner [Newell and H. Simon, 1972]. GPS planner was able to *use* an abstraction hierarchy, but it was a human expert who *determined* the importance of literals. ABSTRIPS planner [Sacerdoti, 1974] was the first attempt to automate the formation of abstraction spaces, but this system needed a human expert to find some outline of a hierarchy, and thus only partially automated the process.

Knoblock in 1990 implemented the abstraction learner ALPINE that completely automates the formation of abstraction hierarchies [Knoblock, 1991a]. ALPINE produces useful abstraction hierarchies in a number of problem domains. To formalize his method, Knoblock introduced the notion of *ordered* abstraction hierarchies that captures the intuition behind "good" abstraction hierarchies. However, hierarchies produced by ALPINE often contain too few levels of abstraction.

In this thesis we continue the work of Knoblock and present a generalization of his methods that allows us to generate *finer-grained* hierarchies, that is hierarchies with more levels of abstraction.

We approach the task from two different directions. The first approach considers the notion of *justified* plans — plans without useless operators. We first introduce several kinds of justified plans. Then we present different kind of *semi-ordered* abstraction hierarchies, which are a generalization of Knoblock's ordered hierarchies, and show how to adapt Knoblock's planner to the use of semi-ordered hierarchies without increasing the size of the search space. We present *necessary and sufficient* conditions for an abstraction hierarchy to be semi-ordered, and demonstrate that these conditions are less restrictive then the conditions used in ALPINE, and therefore they enable us to generate finer-grained hierarchies.

The second approach is based on the notion of *primary effects* of operators. We describe a method to determine such effects, and introduce the notion of *primary-effect restricted* planners, which apply an operator only for the sake of its primary effects. For example, suppose you are boiling water in a microwave. The primary effect of this action is to obtain a cup of hot water — this is your goal. *Side effects* are heating the room, spending electricity, and so on. A primary-effect restricted planner never uses an operator to achieve its side effect. Such a planner would not suggest to boil water in order to heat the room. We show that the primary-effect restricted planning allows us to generate finer-grained ordered abstraction hierarchies than those generated by ALPINE.

Finally, we consider Knoblock's method of generating problem-specific hierarchies, present a further formalization of this method, and adapt it to semi-ordered and primary-effect restricted hierarchies.

1.2 Outline of the thesis

The reader's guide in Figure 1.2 presents the order in which you may read the thesis.

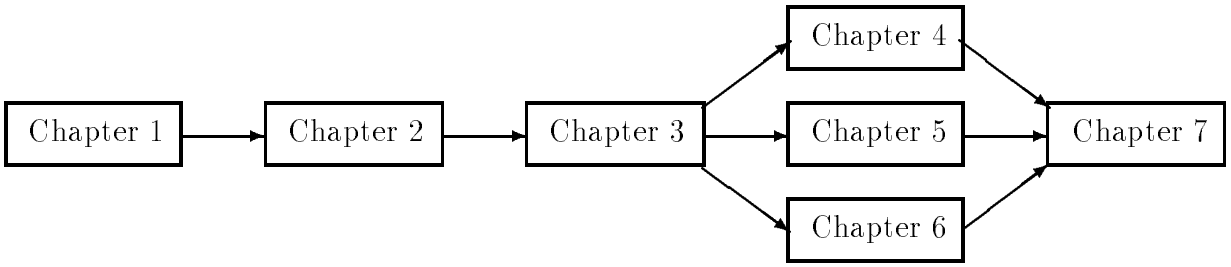


Figure 1.1: Reader's guide

In Chapter 2 we present a formal description of problems domain and introduce definitions and notation that we are going to use. (The summary of the notation is presented in Appendix A). Then we prove several basic facts about planning domains. The second part of the chapter describes the data structures for representing plans and abstraction hierarchies in computer memory, and basic algorithms to work with these data structures.

In chapter 3 we introduce four kinds of plan justifications. We compare different kinds of justifications in terms of the lengths of justified plans and running times necessary to find justifications of a given plan. We present algorithms to find three of them and prove that the problem to find the fourth, "best possible" justification of a plan is NP-complete.

Chapter 4 describes several kinds of semi-ordered hierarchies, some of which are finer-grained than others. We present sufficient and necessary conditions for a hierarchy to be semi-ordered, and an algorithm based on these conditions for generating an abstraction hierarchy. Then we describe an algorithm based on a learning technique that helps us further increase the number of levels.

Chapter 5 describes a method to determine primary effects of operators. This method leads to the notion of primary-effect restricted abstraction hierarchies that enables us to improve efficiency of planning in many domains. We present sufficient and necessary conditions of *the completeness property* of primary-effect restricted hierarchies, which ensures the completeness of the planning algorithm.

Chapter 6 introduces the notion of a *goal-specific domain*, which allows us to further increase the number of abstraction levels by generalizing Knoblock's method [Knoblock, 1991a] of building a hierarchy tailored to a specific goal[1].

Finally, a summary of the thesis is given in Chapter 7, along with a discussion of the work that has yet to be done.

[1]The term "goal-specific" in this thesis corresponds to Knoblock's term "problem-specific". We prefer the term "goal-specific" because a modification of an abstraction hierarchy depends only on a goal of the plan.

Chapter 2

Definitions, notation, and basic algorithms

In this chapter we introduce definitions and notations of planning domains, and present data structures and basic algorithms used in the subsequent chapters. To help the reader to understand the material easier, we present the summary of notations in Appendix A.

2.1 A formal description of a planning domain

We consider a model of the world described by a finite set of variables $\mathcal{X}=\{x_1, x_2, \ldots, x_n\}$. Each variable x can accept one of several values. The set of all values that x can accept is called the *domain* of x and denoted by $D(x)$. We assume that the domain of each variable is finite. To describe a *complete state* S of the world, we specify the values of all variables in the domain. Thus, we pick some value from the domain of x_1, $D(x_1)$, and assign it to x_1, then pick some value from $D(x_2)$ and assign it to x_2, and so on until every variable has some value. Formally, this assignment may be viewed as a function S from the set of domain variables $\mathcal{X}$ into the set of their values, that is into the set $D(x_1) \cup D(x_2) \cup ... \cup D(x_n)$. This function projects each variable x_k into its own domain, $D(x_k)$, which may be represented by the formal expression

$$(\forall x_k \in \mathcal{X})\ S(x_k) \in D(x_k)$$

We write "$(x = v) \in S$" to mean that the value of x is equal to v in the state S. Also, we sometimes write "$S(x)$" to refer to the value of x in the state S: if $(x = v) \in S$ then $S(x) = v$.

As an example, suppose we have a cup, a glass, and a kettle, each of which may contain water. The kettle and the cup may contain either hot or cold water, while water in the glass must not be hot. This domain may be described with three variables:

$$\{\text{Kettle, Cup, Glass}\}$$

The possible values of variables *Kettle* and *Cup* are *Empty*, *Cold-Water*, and *Hot-Water*, while *Glass* may be assigned only two values, *Empty* or *Cold-Water*. Formally, the domains of this three variables may be written as

$$\begin{array}{lcl} D(\text{Kettle}) & = & \{\text{Empty, Cold-Water, Hot-Water}\} \\ D(\text{Cup}) & = & \{\text{Empty, Cold-Water, Hot-Water}\} \\ D(\text{Glass}) & = & \{\text{Empty, Cold-Water}\} \end{array}$$

The state S_1 in which the cup and glass are empty, and the kettle contains cold water, may be described formally as follows:

$$S_1 = \{(\text{Kettle=Cold-Water}), (\text{Cup=Empty}), (\text{Glass=Empty})\}$$

The value of *Kettle* in the state S is *Cold-Water*, which may be written as

$$S_1(\text{Kettle}) = \text{Cold-Water}$$

If we know the values of some variables, and do not know the others, we may specify only the known values. Such a specification is called a *partial state* of the world. Formally, a partial state S is a *partial function* from the set of variables $\mathcal{X}$ into set of the values $D(x_1) \cup D(x_2) \cup ... \cup D(x_n)$, such that each variable x_i is mapped either into its own domain $D(x_i)$, or not mapped at all. We write $x \sqsubset S$ to mean that the value of a variable x is specified in a partial state S. For example, if the kettle contains cold water, the cup is empty, and the content of the glass is unknown, we may express this information as a partial state

$$S_2 = \{(\text{Kettle=Cold-Water}), (\text{Cup=Empty})\}$$

The values of variables *Kettle* and *Cup* are specified in this state, while the value of *Glass* is not specified. This may be formally written as

$$\begin{array}{l} \text{Kettle} \sqsubset S_2 \\ \text{Cup} \sqsubset S_2 \\ \text{Glass} \not\sqsubset S_2 \end{array}$$

We use the word *state* without any adjective if we do not know whether the state is partial or complete. We denote the number of variables whose values are specified in the state S by $|S|$. Observe that we may receive several different complete states from a partial state of the world by specifying unspecified variables. Thus, every partial state corresponds to several complete states.

An *operator* α is defined by an ordered pair ($Pre(\alpha)$,$Eff(\alpha)$), where $Eff(\alpha)$ is a set of values of the form $(x = v)$, that is the definition of $Eff(\alpha)$ is the same as the definition of a state, and $Pre(\alpha)$ may contain specifications of variable values of the two forms: $(x = v)$ and $(x \neq v)$, where the latter means that the value of x is not equal to v. $Pre(\alpha)$ is called the *set of preconditions* of an operator α, and $Eff(\alpha)$ is called the *set of effects* of α. We require that the set of effects $Eff(\alpha)$ is *not empty*, and its intersection with the set of preconditions is empty, that is the same value $(x = v)$ cannot be both a precondition and an effect of an operator:

$Eff(\alpha) \neq \emptyset$, and
if $(x = v) \in Eff(\alpha)$ then $(x = v) \notin Pre(\alpha)$

operator α	**preconditions** $Pre(\alpha)$	**effects** $Eff(\alpha)$
Fill-Kettle	(Kettle=Empty)	(Kettle=Cold-Water)
Boil-Kettle	(Kettle≠Empty)	(Kettle=Hot-Water)
Fill-Cup-Hot	(Kettle=Hot-Water), (Cup=Empty)	(Cup=Hot-Water)
Fill-Cup-Cold	(Kettle=Cold-Water), (Cup=Empty)	(Cup=Cold-Water)
Empty-Cup	(Cup≠Empty)	(Cup=Empty)
Heat-Cup	(Cup≠Empty)	(Cup=Hot-Water)
Fill-Glass	(Kettle=Cold-Water), (Glass=Empty)	(Glass=Cold-Water)
Empty-Glass	(Glass≠Empty)	(Glass=Empty)

Table 2.1: Operators in the water-boiling domain

Intuitively, the first requirement means that each operator produces some effect, and the second requirement states that an operator does not achieve a value that always holds *before* the execution of the operator.

As an example, we consider our domain with a kettle, a cup, and a glass. Suppose that we may fill the kettle with cold water, and that we may pour water from the kettle into either the cup or glass. (We assume that the kettle is large and does not become empty when we pour water from it into the cup or glass, even if we do it repeatedly.) We may empty the cup or the glass by pouring water into a sink. Also, we may boil water in the kettle by putting the kettle onto a stove, and we may heat water in the cup by putting the cup into a microwave. The formal description of all these operations is shown in Table 2.1. For example, the precondition of an operator *Fill-Kettle* states that the kettle must be empty before we fill it, and after an execution of this operator the kettle contains cold water. The operator *Fill-Glass* has two preconditions: glass must be empty and kettle must contain cold water. If the preconditions are satisfied, we may execute the operator and obtain a glass of cold water.

We denote the number of preconditions of an operator α by $|Pre(\alpha)|$, and the number of its effects by $|Eff(\alpha)|$. We write $Pre(\alpha) \subseteq S$ if for every $(x_1 = v_1) \in Pre(\alpha)$, x_1 has the value v_1 in the state S, and for every $(x_2 \neq v_2) \in Pre(\alpha)$, x_2 is specified in S and its value is different from v_2 in S:

$$Pre(\alpha) \subseteq S \text{ if and only if } (\forall x \in \mathcal{X}) \begin{cases} \text{if } (x = v) \in Pre(\alpha) \text{ then } (x = v) \in S \\ \text{if } (x \neq v) \in Pre(\alpha) \text{ then } (x = v_1) \in S, \text{ where } v_1 \neq v \end{cases}$$

If $Pre(\alpha) \subseteq S$, we say that the preconditions of the operator α_1 are satisfied in S, or that α is *legal* in the state S. For example, the preconditions of the operator *Heat-Cup* are satisfied in the state

$$S = \{\text{(Cup=Cold-Water), (Glass=Empty)}\}$$

since the cup is not empty in this state. From now on, in all theorems, algorithms, and almost all definitions we will consider only one kind of preconditions: $(x = v)$. One may check that all our proofs are also correct for the preconditions of the form $(x \neq v)$. We do

not consider this kind of preconditions in ordered to avoid repeating almost the same thing twice.

If α is legal in S, we can *apply* this operator. Its application produces a new state $\alpha(S)$, where all variables whose values are specified by *Eff*(α) have received these new values, and all other variables have the same values as in S. Formally,

$(x = v) \in \alpha(S)$ if and only if
- $(x = v) \in$ *Eff*(α), or
- $(x = v) \in S$ and $x \not\sqsubset$ *Eff*(α)

If $(x = v) \in$ *Eff*(α), we say that the operator α *achieves* or *establishes* the value v of x, or, shortly, achieves $(x = v)$. If $x \sqsubset$ *Eff*(α), we say that α *changes* x. If α establishes $(x = v)$, and v' is a value of x different from v, that is if $v \neq v'$, we sometimes say that α establishes $(x \neq v')$, or that $(x \neq v')$ is an effect of α.

The set of *outcomes*, *Out*(α), of an operator α is defined as the set of all its effects together with the set of the preconditions which are not changed by the effects of the operator:

$$Out(\alpha) = Eff(\alpha) \cup \{(x = v) \in Pre(\alpha) \mid x \not\sqsubset Eff(\alpha)\}$$

Observe that if α is legal in some state S, then after an execution of α all its outcomes hold:

$$\text{if } Pre(\alpha) \subseteq S \text{ then } Out(\alpha) \subseteq \alpha(S)$$

Intuitively, the outcomes of an operator are the literals that always hold after a legal execution of an operator. If $(x = v)$ is an outcome of α, and v' is a value of x different from v, we say that the operator α *negates* $(x = v')$, or that $(x \neq v')$ is an outcome of α. This means that the value of x after a legal execution of α is always different from v'.

For example, consider an operator *Fill-Glass*. The only effect of the operator is cold water in the glass. However, we may deduce more information about the state of the world after the execution of this operator. Since we may fill the glass only if water in the kettle is cold, and the operator does not change the temperature of water and does not make kettle empty, we conclude that the kettle must contain cold water after a legal execution of *Fill-Glass*. Thus, this operator has two outcomes:

$$Out(\text{Fill-Glass}) = \{(\text{Glass=Cold-Water}), (\text{Kettle} = \text{Cold-Water})\}$$

The set of all operators in a problem domain is denoted by O. We assume that O is finite. For a given value v of x, we denote the set of all operators that achieve $(x = v)$ by $O_{(x=v)}$, that is

$$O_{(x=v)} = \{\alpha \in O \mid (x = v) \in Eff(\alpha)\}$$

For a given variable x, we denote the set of all literals that change x by O_x, that is

$$O_x = \{\alpha \in O \mid x \sqsubset Eff(\alpha)\}$$

We say that a value v of x is *achievable* if $O_{(x=v)} \neq \emptyset$, i.e. there is an operator that achieves $(x = v)$. We say that a variable x is *changeable* if $O_x \neq \emptyset$, i.e. there is an operator that changes x. For each variable x we denote the set of its achievable values by $D_a(x)$, and the

set of its unachievable values by $D_u(x)$. Similarly, we denote the set of changeable variables by $\mathcal{X}_c$, and the set of unchangeable variables by $\mathcal{X}_u$.

A *linear plan* is a triple $(S_0, S_g, \overline{\Pi})$, where S_0 is an *initial state*, S_g is a *goal state*, and $\overline{\Pi} = (\alpha_1, \ldots, \alpha_n)$ is a finite sequence of operators. Sometimes we use our terminology loosely and call "plan" a sequence $\overline{\Pi}$ alone. The linear plan may be executed by applying operators in order: first we apply α_1 to S_0 and receive a new state S_1, then we apply α_2 and receive the next state, S_2, and so on until we have applied α_n. Formally, the m-th state of the plan is defined as follows

$$(\forall m \in [1..n])\ S_m = \alpha_m(S_{m-1}) = \alpha_m(\alpha_{m-1}(\ldots(\alpha_1(S_0))\ldots))$$

The final state of the plan, which is the result of the plan execution, is denoted by $\overline{\Pi}(S_0)$:

$$\overline{\Pi}(S_0) = S_n = \alpha_n(\alpha_{n-1}(\ldots(\alpha_1(S_0))\ldots))$$

We say that a plan $(S_0, S_g, \overline{\Pi})$ is *legal* if the preconditions of every operator are satisfied before the execution of the operator:

$$(\forall m \in [1..n])\ Pre(\alpha_m) \subseteq S_{m-1}$$

We say that a plan is *correct* if it is legal and achieves the goal, that is $S_g \subseteq \overline{\Pi}(S_0)$.

Observe that the value of a variable x is specified in some intermediate state S_m of a linear plan if and only if it is specified in the initial state or achieved by some operator preceding S_m. We state this result as a lemma.

Lemma 2.1 *Let $\overline{\Pi} = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ be a linear plan with an initial state S_0. Then for any $m \in [0..n]$, $x \sqsubset S_m$ if and only if*

(1) $x \sqsubset S_0$, or

(2) $(\exists k \in [1..m])$ such that $x \sqsubset \mathrm{Eff}(\alpha_k)$

Proof. Let us use the notation $dom(S)$ to denote the set of the variables whose values are specified in a state S, and $dom(Eff(\alpha))$ to denote the set of variables changed by the operator α. Then it immediately follows from the definition of $\alpha(S)$ that

$$dom(\alpha(S)) = dom(S) \cup dom(Eff(\alpha))$$

By applying this equality to the definition of S_k, we receive

$$dom(S_k) = dom(S_0) \cup dom(Eff(\alpha_1)) \cup \ldots \cup dom(Eff(\alpha_k))$$

This may be rewritten as

$$(x \sqsubset S_k) \iff (x \sqsubset S_0) \vee (x \sqsubset Eff(\alpha_1)) \vee \ldots \vee (x \sqsubset Eff(\alpha_k)$$

which proves the lemma. □

A *nonlinear plan* is a triple (S_0, S_g, Π), where S_0 is an initial state, S_g is a goal state, and Π is a set of operators $\{\alpha_1, \ldots, \alpha_n\}$ with a partial order $\prec_\Pi$ on it. (The subscript to $\prec$ will be

dropped if the intended plan is unambiguously specified.) This partial order represents the time-precedence relation between operators: $\alpha_1 \prec \alpha_2$ means that α_1 must be executed before α_2. A sequence $\overline{\Pi} = (\alpha_{k_1}, \ldots, \alpha_{k_n})$ is a *linearization* of Π if it contains all the operators of Π, and the order defined by $\prec$ is not violated, that is for any α_i and α_j, if $\alpha_i \prec \alpha_j$, then α_i occurs before α_j in $\overline{\Pi}$. A linear plan $(S_0, S_g, \overline{\Pi})$ is a linearization of a nonlinear plan (S_0, S_g, Π) if $\overline{\Pi}$ is a linearization of Π. A nonlinear plan is legal if all its linearizations are legal, and it is correct if all its linearizations are correct. The number of operators in Π is denoted by $|\Pi|$. Also, for a given plan (S_0, S_g, Π) we use the letter P to denote the sum of the number of preconditions of all operators in Π plus the number of goal values, and E to denote the number of effects of all operators:

$P = \sum_{\alpha \in O} |Pre(\alpha)| + |S_g|$, and
$E = \sum_{\alpha \in O} |Eff(\alpha)|$

We say that an operator α_1 *necessarily* precedes an operator α_2 in a nonlinear plan Π if α_1 precedes α_2 in all linearizations of Π. An operator α_1 *possibly* precedes α_2 if α_1 precedes α_2 in at least one linearization of Π. It is easy to check [Chapman, 1987] that

- α_1 is necessarily before α_2 if and only if $\alpha_1 \prec \alpha_2$, and
- α_1 is possibly before α_2 if only if $\neg(\alpha_2 \prec \alpha_1)$

Throughout the remainder of the thesis all plans are assumed to be nonlinear unless otherwise specified.

A plan (S_0, S_g, Π) is a *subplan* of (S_0, S_g, Π') if

$\forall \alpha_1, \alpha_2 \in \Pi$
(1) $\alpha_1, \alpha_2 \in \Pi'$, and
(2) $\alpha_1 \prec_{\Pi} \alpha_2 \Leftrightarrow \alpha_1 \prec_{\Pi'} \alpha_2$

Intuitively, a subplan may be received from a plan by removing several operators, and preserving the order of the remaining operators. If a subplan Π' of Π contains less operators than Π, that is Π' is *not* the plan Π itself, then Π' is called a *proper* subplan of Π. If (S_0, S_g, Π') is a legal plan that achieves S_g, we say it is a *correct subplan* of (S_0, S_g, Π). The following lemma states a simple, but important property of subplans, which we use in proofs of some theorems.

Lemma 2.2 *Let Π be a subplan of a nonlinear plan Π', and $\overline{\Pi}$ be some linearization of Π. Then there exists a linearization $\overline{\Pi}'$ of Π' such that $\overline{\Pi}$ is a subplan of $\overline{\Pi}'$.*

Intuitively, we take the linear plan $\overline{\Pi}$ and insert into it the operators of Π' which are not in $\overline{\Pi}$. If we add all these operators without violating the time-precedence relation of Π', then the resulting linear plan $\overline{\Pi}'$ is a linearization of Π', and $\overline{\Pi}$ is a subplan of $\overline{\Pi}'$. The following proof shows that the operators of Π' which are not in $\overline{\Pi}$ indeed may be inserted into $\overline{\Pi}$ without violating their order in Π'.

Proof. First assume that Π has one operator less than Π', that is

$$|\Pi| = |\Pi'| - 1$$

Let us denote operators of Π, according to their order in $\overline{\Pi}$, as follows:

$$\overline{\Pi} = (\alpha_1, \alpha_2, \ldots, \alpha_n)$$

We denote the operator of Π' that is missing in Π by α.

Let $\alpha_{k_1}, \alpha_{k_2}, \ldots, \alpha_{k_i}$ be the set of operators that necessarily precede α according to the constraints of the plan Π', and $\alpha_{m_1}, \alpha_{m_2}, \ldots, \alpha_{m_j}$ be the set of operators that must be executed after α:

$$\alpha_{k_1}, \alpha_{k_2}, \ldots, \alpha_{k_i} \prec_{\Pi'} \alpha \prec_{\Pi'} \alpha_{m_1}, \alpha_{m_2}, \ldots, \alpha_{m_j}$$

W.l.o.g. we may assume that

$k_1 < k_2 < \ldots < k_i$, and
$m_1 < m_2 < \ldots < m_j$

that is α_{k_1} is executed in $\overline{\Pi}$ before α_{k_2}, α_{k_2} before α_{k_3}, and so on. Since $\alpha_{k_i} \prec_{\Pi'} \alpha \prec_{\Pi'} \alpha_{m_1}$, by transitivity of the time-precedence relation we conclude that $\alpha_{k_i} \prec_{\Pi'} \alpha_{m_1}$, and therefore by the definition of subplans, $\alpha_{k_i} \prec_{\Pi} \alpha_{m_1}$. Then, by the definition of a linearization, α_{k_i} occurs in $\overline{\Pi}$ before α_{m_1}, that is $k_i < m_1$.

Let plan $\overline{\Pi}'$ be obtained by inserting α into $\overline{\Pi}$ immediately after α_{k_i}. Then α is located in $\overline{\Pi}'$ after $\alpha_{k_1}, \alpha_{k_2}, \ldots, \alpha_{k_i}$, and before $\alpha_{m_1}, \alpha_{m_2}, \ldots, \alpha_{m_j}$. Thus, the location of α satisfies the time-precedence relation of the plan Π', and therefore $\overline{\Pi}'$ is a linearization of Π'. Clearly, $\overline{\Pi}$ is a subplan of $\overline{\Pi}'$, and thus a required linearization of Π' is found.

Now assume that more than one operator of Π' is missing in Π, that is

$$|\Pi| = |\Pi'| - k, \text{ where } k > 1$$

Then we may find a sequence of plans $\Pi_1, \Pi_2, \ldots, \Pi_{k-1}$ such that Π_1 is obtained from Π' by removing one operator, Π_2 is obtained from Π_2 by removing one more operator, and so on till Π_{k-1}, and finally Π is obtained from Π_{k-1} also by removing one operator. Then, according to the first part of the proof, there exists a sequence of linear plans $\overline{\Pi}', \overline{\Pi}_1, \overline{\Pi}_2, \ldots, \overline{\Pi}_{k-1}$, which are linearizations of respectively $\Pi', \Pi_1, \Pi_2, \ldots, \Pi_{k-1}$, such that $\overline{\Pi}$ is a subplan of $\overline{\Pi}_{k-1}$, $\overline{\Pi}_{k-1}$ is a subplan of $\overline{\Pi}_{k-2}$, ..., $\overline{\Pi}_1$ is a subplan of $\overline{\Pi}'$. Then $\overline{\Pi}$ is a subplan of $\overline{\Pi}'$, and thus $\overline{\Pi}'$ is a required linearization of Π. □

Intuitively we expect that if we have built a *correct* plan based on some partial knowledge of an initial state of the world, then any additional knowledge cannot make the plan incorrect. In other words, if a plan is correct in some partial state S_0, it has to be correct in any complete state received from S_0 by specifying the unspecified variables. For example, suppose you need a cup of hot water, and you have a kettle with cold water, an empty cup, and a stove. This information is enough to find the following correct plan for achieving your goal:

1. Boil water by putting the kettle onto the stove.
2. Pour water into the cup.

Any additional information, such as knowledge of the initial temperature of the water or whether the glass is empty or full (recall that we have a glass in our domain), cannot make your plan incorrect. The following lemma shows that this property indeed holds, and thus our formal definition of the correctness corresponds to human intuition.

Lemma 2.3 *Let (S_0, S_g, Π) be a correct plan, and $S_0 \subseteq S'_0$. Then the plan (S'_0, S_g, Π) is also correct.*

Proof. To show that (S'_0, S_g, Π) is correct, we need to show that any of its linearizations is correct. So consider an arbitrary linearization $\overline{\Pi} = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ of Π. The plan $(S_0, S_g, \overline{\Pi})$ is correct as a linearization of a correct plan. We denote the intermediate states of $(S_0, S_g, \overline{\Pi})$ by $S_1, S_2, \ldots, S_n$, and the intermediate states of $(S'_0, S_g, \overline{\Pi})$ by $S'_1, S'_2, \ldots, S'_n$. Since $(S_0, S_g, \overline{\Pi})$ is correct, by the definition of the correctness we have:

$$S_g \in S_n \text{ and } (\forall k \in [1..n])\ \mathit{Pre}(\alpha_k) \subseteq S_{k-1} \qquad (1)$$

To prove that $(S'_0, S_g, \overline{\Pi})$ is correct, we need to show that

$$S_g \in S'_n \text{ and } (\forall k \in [1..n])\ \mathit{Pre}(\alpha_k) \subseteq S'_{k-1} \qquad (2)$$

Observe that if we prove that

$$(\forall k \in [0..n])\ S_k \subseteq S'_k \qquad (3)$$

than statement (2) directly follows from statements (1) and (3). Thus, to show that the plan $(S'_0, S_g, \overline{\Pi})$ is correct, it is enough to prove statement (3). We prove it by induction on k.

Base. $S_0 \subseteq S'_0$ by the statement of the theorem.

Step. Assume that for some k, $S_k \subseteq S'_k$. We need to show that $S_{k+1} \subseteq S'_{k+1}$. By definitions presented in this section and using elementary properties of sets, we have:

$$\begin{array}{ll}
& S_{k+1} \subseteq S'_{k+1} \\
\iff & \alpha_{k+1}(S_k) \subseteq \alpha_{k+1}(S'_k) \\
\iff & (\mathit{Eff}(\alpha_{k+1}) \cup \{(x = v) \in S_k \mid x \not\sqsubset \mathit{Eff}(\alpha_{k+1})\}) \\
& \qquad \subseteq (\mathit{Eff}(\alpha_{k+1}) \cup \{(x = v) \in S'_k \mid x \not\sqsubset \mathit{Eff}(\alpha_{k+1})\}) \\
\Longleftarrow & \{(x = v) \in S_k \mid x \not\sqsubset \mathit{Eff}(\alpha_{k+1})\} \subseteq \{(x = v) \in S'_k \mid x \not\sqsubset \mathit{Eff}(\alpha_{k+1})\} \\
\iff & (\forall x)\ \not\sqsubset \mathit{Eff}(\alpha_{k+1})) \text{ if } (x = v) \in S_k \text{ then } (x = v) \in S'_k \\
\Longleftarrow & (\forall x) \text{ if } (x = v) \in S_k \text{ then } (x = v) \in S'_k \\
\iff & S_k \subseteq S'_k
\end{array}$$

Thus, we have shown that $S_{k+1} \subseteq S'_{k+1}$ follows from $S_k \subseteq S'_k$, as desired. □

2.2 Example of a Planning Domain (Tower of Hanoi)

In this example we describe a generalized tower of Hanoi puzzle, which is then used in the following chapters to illustrate techniques for hierarchical problem solving. The puzzle consists of several (not necessarily three) pegs with various-sized disks on them. We consider an instance of the puzzle with three disks, small, medium, and large, and four pegs, denoted by 1, 2, 3, and 4. We may move a disk from one peg onto another, one disk at a time. The constraints are that a disk can only be moved if it is above all other disks on a peg, and a

Figure 2.1: The tower of Hanoi with four pegs

operator α	**preconditions** $Pre(\alpha)$	**effects** $Eff(\alpha)$	**outcomes** $Out(\alpha)$
$Move_S$(a,b)	($Where_S$=a)	($Where_S$=b)	($Where_S$=b)
$Move_M$(a,b)	($Where_M$=a) ($Where_S$≠a), ($Where_S$≠b)	($Where_M$=b)	($Where_M$=b) ($Where_S$≠a), ($Where_S$≠b)
$Move_L$(a,b)	($Where_L$=a) ($Where_S$≠a), ($Where_S$≠b) ($Where_M$≠a), ($Where_M$≠b)	($Where_L$=b)	($Where_L$=b) ($Where_S$≠a), ($Where_S$≠b) ($Where_M$≠a), ($Where_M$≠b)

Table 2.2: The operator types in our tower of Hanoi domain

larger disk can never be placed on a smaller one. Figure 2.1 shows the tower of Hanoi puzzle with three disks and four pegs.

We may describe the state of the world in this tower of Hanoi puzzle with three variables: $Where_S$, $Where_M$, and $Where_L$, which represent the positions of respectively the small, medium, and large disks. The domain of all three variables is the set of pegs, $\{1, 2, 3, 4\}$. For example, $(Where_S = 1)$ means that the small disk is on peg 1. The state of the world shown on the Figure 2.1 may be formally described as

$$S = \{(Where_S = 1), (Where_M = 1), (Where_L = 1)\}$$

The formal description of the operators is given in Table 2.2. Letters a and b in the table denote arbitrary pegs. For example, $Move_S(a, b)$ denotes an operator that moves the small disk from some peg a to some other peg b. We receive a particular operator by substituting some specific pegs (1, 2, 3, or 4) for a and b. Thus, the notation $Move_S(a, b)$ presents not a specific operator, but an *operator type*. Sometimes we use the terminology loosely and call $Move_S(a, b)$ an operator. The additional column in the table shows the outcomes of each operator,

$$Out(\alpha) = Eff(\alpha) \cup \{(x = v) \in Pre(\alpha) \mid x \not\sqsubset Eff(\alpha)\}$$

which always hold after a legal execution of the operator.

Let us find a plan for achieving the state shown in Figure 2.2, which is formally described as

$$S_g = \{(Where_S = 1), (Where_M = 2), (Where_L = 2)\}$$

A linear plan that achieves it is

$$\overline{\Pi} = (Move_S(1, 3);\ Move_M(1, 4);\ Move_L(1, 2);\ Move_M(4, 2);\ Move_S(3, 1))$$

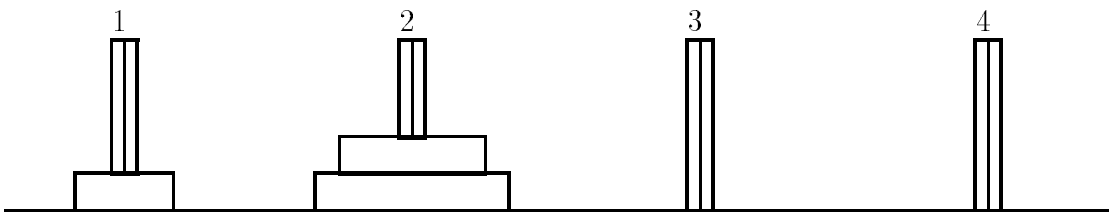


Figure 2.2: The goal of our plan

It is easy to see that the order of the last two operators does not matter. So, we may use a nonlinear plan, where this order is not specified:

$$\longrightarrow \mathit{Move_S}(1,3) \longrightarrow \mathit{Move_M}(1,4) \longrightarrow \mathit{Move_L}(1,2) \begin{array}{c} \nearrow \\ \searrow \end{array} \begin{array}{c} \mathit{Move_S}(4,2) \\ \\ \mathit{Move_M}(3,1) \end{array} \begin{array}{c} \searrow \\ \nearrow \end{array} \longrightarrow$$

2.3 Literal-Representation vs. Variable-Representation

In the previous section we showed how to describe states of the world as sets of values of domain variables. This representation of states is called the *variable-representation*. In this section we consider a slightly different method to describe states of the world in the problem domain, called the *literal-representation*. The literal-representation has been used in most research in classical planning. Then we show how these two representations can be mapped into each other, and emphasize some advantages of the variable-representation.

To obtain the literal-representation of a problem domain, we introduce a finite set of *literals* $\mathcal{L}=\{l_1, l_2, \ldots, l_n\}$ instead of a set of variables $\mathcal{X}$. For each literal l in $\mathcal{L}$, the negation of l, $\neg l$, also belongs to $\mathcal{L}$. At any given moment in time each literal l is either *True* or *False*, but not both. If a literal l is *True*, then its negation, $\neg l$, is *False*, and vice versa.

A *complete state* S of the world is such a set of literals that for every literal l from $\mathcal{L}$, S contains either l or $\neg l$, but not both:

$$(\forall l \in \mathcal{L})\ l \in S \ \text{ or } \ \neg l \in S, \text{ but not both}$$

The literals contained in the complete state S are assumed to be *True*, while all other literals are *False*. Thus, the complete state specifies the values of all literals in the problem domain.

A *partial state of the world* S is a set of literals that cannot contain a literal and its negation at the same time, and that does not specify the truth value of at least one literal:

$$\begin{array}{ll} (1) & (\forall l \in \mathcal{L})\ l \notin S \ \text{ or } \ \neg l \notin S, \text{ and} \\ (2) & (\exists l \in \mathcal{L})\ l \notin S \ \text{ and } \ \neg l \notin S \end{array}$$

Thus, S may contain some literal l, or its negation, $\neg l$, or neither, but S cannot contain both l and $\neg l$. If $l \in S$, the truth value of l is assumed to be *True*, and if $\neg l \in S$, the value

operator α	**preconditions** $Pre(\alpha)$	**effects** $Eff(\alpha)$
Move_S(a,b)	S_on(a)	S_on(b), ¬S_on(a)
Move_M(a,b)	M_on(a) ¬S_on(a), ¬S_on(b)	M_on(b), ¬M_on(a)
Move_L(a,b)	L_on(a) ¬S_on(a), ¬S_on(b) ¬M_on(a), ¬M_on(b)	L_on(b), ¬L_on(a)

Table 2.3: The operator types in the literal-represented tower of Hanoi

of l is *False*. If neither l nor $\neg l$ belongs to S, the truth value of l is unknown. If S is a partial state, then there is at least one literal whose value is unknown.

The preconditions $Pre(\alpha)$ and effects $Eff(\alpha)$ of an operator α are defined the same way as partial states: they are sets of literals that do not contain a literal and its negation at the same time. Also, $Eff(\alpha)$ must not be empty, $Eff(\alpha) \neq \emptyset$, and the same literal cannot be both in the preconditions and the effects of an operator, $Eff(\alpha) \cap Pre(\alpha) = \emptyset$. An operator α is legal in some state S if $Eff(\alpha) \subseteq S$. The application of α to S produces a new state $\alpha(S)$, where all literals from $Eff(\alpha)$ hold, and all literals that do not conflict with $Eff(\alpha)$ are left unchanged:

$$\alpha(S) = Eff(\alpha) \cup \{l \in S \mid \neg l \notin Eff(\alpha)\}$$

E.g., if l_1, l_2, and l_3 are literals in our domain, $S = \{l_1, \neg l_2, l_3\}$, and $Eff(\alpha) = \{\neg l_1\}$, then applying α to S creates a new state $\{\neg l_1, \neg l_2, l_3\}$.

Example (Literal-representation of the tower of Hanoi)
In this example we describe the literal-representation of the tower of Hanoi domain. We again consider the tower of Hanoi with three disks and four pegs. States of the world in this domain may be described with twenty-four literals:

$S_on(1)$	$\neg S_on(1)$	$M_on(1)$	$\neg M_on(1)$	$L_on(1)$	$\neg L_on(1)$
$S_on(2)$	$\neg S_on(2)$	$M_on(2)$	$\neg M_on(2)$	$L_on(2)$	$\neg L_on(2)$
$S_on(3)$	$\neg S_on(3)$	$M_on(3)$	$\neg M_on(3)$	$L_on(3)$	$\neg L_on(3)$
$S_on(4)$	$\neg S_on(4)$	$M_on(4)$	$\neg M_on(4)$	$L_on(4)$	$\neg L_on(4)$

where, for example, $S_on(1)$ means "the small disk in on peg 1", and $\neg L_on(2)$ stands for "the large disk is not on peg 2". For convenience, we combine this twenty-four literals into six literal types:

$S_on(a)$	$M_on(a)$	$L_on(a)$
$\neg S_on(a)$	$\neg M_on(a)$	$\neg L_on(a)$

where a stands for an arbitrary peg. The formal description of operator types is given in Table 2.3.

Suppose we wish to find a plan with the same initial and goal states as in the previous example (see Figure 2.2). The literal-representation of these initial and goal states are

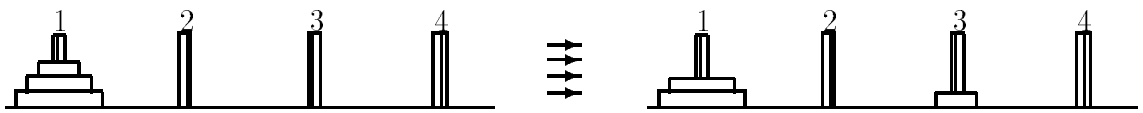


Figure 2.3: The application of an operator *Move_S*(1,3)

$$S_0 = \{S_on(1), M_on(1), L_on(1)\}$$
$$S_g = \{S_on(1), M_on(2), L_on(2)\}$$

Of course, the goal may be achieved by the same plan as in the variable-represented domain:

$$\overline{\Pi} = (Move_S(1,3),\ Move_M(1,4),\ Move_L(1,2),\ Move_M(4,2),\ Move_S(3,1))$$

To get an intuitive idea how we describe the application of an operator to a state, let us consider the application of the first operator of the plan, *Move_S*(1,3), to the initial state (see Figure 2.3). The precondition of this operator, $S_on(1)$, holds in the initial state, and therefore the operator is legal. Both effects of the operator, $S_on(3)$ and $\neg S_on(1)$, must hold in the resulting state S_1. The literals $M_on(1)$ and $L_on(1)$ of the initial state does not conflict with the newly achieved literals, and so they still hold in S_1. On the other hand, the literal $S_on(1)$ of the initial state conflicts with a newly achieved literal $\neg S_on(1)$, and therefore this literal does not belong to the resulting state S_1. Thus,

$$S_1 = \{S_on(3), \neg S_on(1), M_on(1), L_on(1)\}$$

Similarly, we may describe the execution of the remaining operators of $\overline{\Pi}$ and show that this plan achieves the goal. □

To convert a literal-representation into a variable-representation, we introduce a variable L for each pair of literals l and $\neg l$ from the literal set $\mathcal{L}$. The variable L may have one of two values, *True* or *False*. Then we may replace the literal l with the expression $(L = True)$, and the literal $\neg l$ with the expression $(L = False)$. After we replace all literals by corresponding expressions, we receive a variable-representation of the problem domain, and it is straightforward to check that this representation is equivalent to the original literal-representation.

On the other hand, a variable-representation of a problem domain always may be converted into a literal-representation. To make such a conversion, we replace every domain variable $x \in \mathcal{X}$, whose possible values are $D(x) = \{v_1, v_2, \ldots, v_m\}$, with $(2 \cdot m)$ literals: $x(v_1), x(v_2), \ldots, x(v_m)$, and $\neg x(v_1), \neg x(v_2), \ldots, \neg x(v_m)$. Thus, we define the set of all literals $\mathcal{L}$ as follows:

$$\mathcal{L} = \{x(v), \neg x(v) \mid x \in \mathcal{X} \text{ and } v \in D(x)\}$$

We map each variable-represented state S_v into a literal-represented state S_l by the following rule:

(1) $x(v) \in S_l$ if and only if $(x = v) \in S_v$
(2) $\neg x(v) \in S_l$ if and only if $(x \neq v) \in S_v$

(Recall that $(x \neq v) \in S_v$ means that the value of x is different from v, that is for some $v' \neq v$, $(x = v') \in S_v$.) We use the same rules (1) and (2) to map the variable-represented preconditions $Pre(\alpha_v)$ and effects $Eff(\alpha_v)$ of every operator α_v into their literal-represented equivalents $Pre(\alpha_l)$ and $Eff(\alpha_l)$. The following lemma shows that the initial variable-representation and the resulting literal-representation describe the same problem domain.

Lemma 2.4 *For every state S and every operator α,*
- $\mathrm{Pre}(\alpha_v) \subseteq S_v$ *if and only if* $\mathrm{Pre}_l(\alpha) \subseteq S_l$
- *mapping the state $\alpha_v(S_v)$ according to rules (1) and (2) produces the state $\alpha_l(S_l)$*

Proof. Assume $Pre(\alpha_v) \subseteq S_v$. To prove that $Pre(\alpha_l) \subseteq S_l$, we need to show that
(i) for all $x(v) \in Pre(\alpha_l)$, $x(v) \in S_l$, and
(ii) for all $\neg x(v) \in Pre(\alpha_l)$, $\neg x(v) \in S_l$
We derive the proof of these two statements as follows.

$$(i) \quad x(v) \in Pre(\alpha_l) \iff (x = v) \in Pre(\alpha_v) \implies (x = v) \in S_v \iff x(v) \in S_l$$

$$(ii) \quad \neg x(v) \in Pre(\alpha_l) \iff (x \neq v) \in Pre(\alpha_v) \implies (x \neq v) \in S_v \iff \neg x(v) \in S_l$$

The reverse direction and the second part of the lemma are proved similarly. □

Observe that while converting the literal-representation into the variable-representation, we replace each literal by *one* value: a literal l is replaced by the value $(L = True)$, and $\neg l$ is replaced by $(L = False)$. On the other hand, when converting in the reverse direction, we need to introduce *two* literals, $v(x)$ and $\neg v(x)$, for each value v of each variable x. Thus, while converting the variable-representation into the literal-representation, we may end up with less compact description of the problem domain than we had before the conversion. Because of this, the variable-representation often allows a more compact description of the problem domain. Also, the variable-representation sometimes allows us to obtain a unary or postunique representation[1] of a problem domain, which may considerably improve the efficiency of planning [Backstrom and Klein, 1991]. Finally, the variable-representation usually better corresponds to the human intuition. For example, in the tower of Hanoi problem it is more natural to think "location of the small disk is on peg 1" than "it is true that the small disk is on peg 1, and it is not true that the small disk is on peg 2"[2]. In the next section we show that the variable-representation has one more advantage: it may help to avoid *initial state axioms*.

2.4 Domain rules

While describing a real-life domain by means of the variable or literal-representation, it often happens that some sets of values do not correspond to any state of the world, or, in other

[1] A planning domain is called *unary* if every operator changes a value of exactly one variable, and it is called *postunique* if no two operators achieve the same value of the same variable, that is for any two distinct operators α_1 and α_2, $Eff(\alpha_1) \cap Eff(\alpha_2) = \emptyset$.

[2] We have to say that this advantage is not universal: there are domains in which literal-representation is closer to the intuition than variable-representation. The blocks world is an example of such a domain.

words, do not make sense. For example, the set of literals $\{S_on(1), S_on(2)\}$ in the literal-represented tower of Hanoi does not describe any real state, since the small disk cannot be on pegs 1 and 2 at the same time. In a formal fashion, this constraint may be written as

$$\neg S_on(1) \vee \neg S_on(2)$$

which means that either the small disk is *not* on peg 1, or it is *not* on peg 2. Such constraint is called a *domain rule*.

We use disjunctive clauses to represent domain rules. Formally, a rule is a clause of the form

$r = l_1 \vee l_2 \vee \ldots \vee l_n$,
where each l_k is either $(x_k = v_k)$ or $(x_k \neq v_k)$
for some $x_k \in \mathcal{X}$ and some $v_k \in D(x_k)$

We say that a complete state S satisfies rules $r_1, r_2, \ldots, r_n$ if, for the values of variables specified in S, $r_1 = r_2 = \ldots = r_n = True$. A partial state satisfies rules $r_1, \ldots, r_n$ if there exists at least one assignment of values of the unspecified variables for which $r_1 = \ldots = r_n = True$. The problem to determine if a partial state satisfies the set of domain rules is generally NP-complete, since it is equivalent to the satisfyability problem for a conjunctive normal form. We say that $R = \{r_1, \ldots, r_n\}$ is a set of domain rules for a problem domain, if every intermediate state of every linear plan in the problem domain satisfies this set of rules.

For example, the constraints in the literal-represented tower of Hanoi may be described with three rule types:

$\neg S_on(a) \vee \neg S_on(b)$
$\neg M_on(a) \vee \neg M_on(b)$
$\neg L_on(a) \vee \neg L_on(b)$

where specific rules may be obtained from each of rule types by replacing a and b with specific distinct pegs. In other words, the rules hold *for all* distinct instances of a and b.

It is easy to see that the states of every plan satisfy a set of domain rules if and only if the initial state of every plan satisfies these rules, and the rules are preserved after the execution of every operator. So, to ensure that some set of rules holds in every state of every plan in a problem domain, we need to check that

1. the rules hold in every possible initial state, and
2. if the rules hold in some state S, and α is an operator legal in S, then all the rules still hold in $\alpha(S)$

This leads us to the notions of initial-state axioms and invariants. The set of *initial-state axioms* A is the set of all rules satisfied by every initial state in the problem domain:

$r \in A$ if and only if, for every plan (S_0, S_g, Π), S_0 satisfies r

The *invariant* I is the set of rules that, once hold in any state, are preserved by any legal operator:

for every operator α and every state S,
if some state S satisfies I and α is legal in S,
then $\alpha(S)$ also satisfies I

Clearly, if an invariant holds in the initial state of a linear plan, than it also holds in all of the following states. Observe that

1. invariants in a problem domain are implicitly defined by the formal description of operators, and

2. domain rules R that hold in every state of problem domain are implicitly defined by a set of initial-state axioms A and invariants.

The first statement means that we may infer from the description of operators which properties of the world the operators preserve. E.g. we may infer that $\neg S_on(a) \vee \neg S_on(b)$ is an invariant in the tower of Hanoi domain by observing that every operator that put the small disk onto some peg b, removes this disk from its previous peg a. This inference is based on a syntactic description of the domain (see Table 2.3), and can be made even if we know nothing about physical properties of a real tower of Hanoi puzzle.

The second statement says that if we know the properties of all possible initial states of the world, and the properties preserved by operators, we may infer the domain rules. For example, if we know that a disk cannot be simultaneously on two different pegs in any initial state, and that every legal operator preserves this property, we conclude that a disk cannot be simultaneously on two different pegs in any intermediate state of any legal plan.

On the other hand, the set of initial-state axioms A must be stated explicitly, since otherwise a planner has no way to determine which initial states are allowed in the problem domain.

For example, consider the variable-representation of the tower of Hanoi with the initial-state axioms:

$$A = \{(Where_S = 1), (Where_M = 1), (Where_L = 1)\}$$

These axioms mean that the only allowed initial state is the state shown in Figure 2.1, where all disks are on peg 1. The set of domain rules in this example is empty, $R = \emptyset$, since *any* state may be achieved by executing some plan.

Now let us consider a less trivial example. Suppose that the tower of Hanoi has only two pegs, 1 and 2, and three disks (see Figure 2.4). Let the initial-state axioms be as follows:

$$A = \{(Where_M = 1), (Where_L = 1)\}$$

These two axioms mean that the medium and large disk must be on peg 1 in the initial state of any plan, while the small disk maybe on any peg. The only possible initial states are the states shown on Figure 2.4. It is easy to check that the only legal operation one can perform with this puzzle is to move the small disk back and forth between the two pegs, but there is no way to put either the medium or large disk onto peg 2. Thus, the domain has two rules, the same as the initial-state axioms:

$$R = \{(Where_M = 1), (Where_L = 1)\}$$

Figure 2.4: Tower of Hanoi with two pegs

which means that the medium and large disk remain on peg 1 in any intermediate state of any legal plan.

Now we present a theorem that describes a connection between invariants and the syntactic description of operators. Let $r = l_1 \vee l_2 \vee \ldots \vee l_n$, where each l_k is either $(x_k = v_k)$ or $(x_k \neq v_k)$, be a rule, and α be an operator such that $\neg l_1, \neg l_2, \ldots, \neg l_k \in \mathit{Eff}(\alpha)$ and $\neg l_{k+1}, \ldots, \neg l_n \notin \mathit{Eff}(\alpha)$, where $0 \leq k \leq n$. That is α negates k literals of r, and for notational convenience we assume, w.l.o.g., that these are the first k literals of r. Let the set of preconditions of α be $\mathit{Pre}(\alpha) = \{l'_1, \ldots, l'_m\}$. We say that α *preserves* r in the case of an invariant I if either

- $k = 0$, that is α negates no literals of r, or
- there exists $l \in r$ such that $l \in \mathit{Eff}(\alpha)$, or
- the following k rules are rules of the invariant I:

$$r_1 = \neg l_1 \vee l_{k+1} \vee l_{k+2} \vee \ldots \vee l_n \vee \neg l'_1 \vee \neg l'_2 \vee \ldots \vee \neg l'_m$$
$$r_2 = \neg l_2 \vee l_{k+1} \vee l_{k+2} \vee \ldots \vee l_n \vee \neg l'_1 \vee \neg l'_2 \vee \ldots \vee \neg l'_m$$
$$\vdots$$
$$r_k = \neg l_k \vee l_{k+1} \vee l_{k+2} \vee \ldots \vee l_n \vee \neg l'_1 \vee \neg l'_2 \vee \ldots \vee \neg l'_m$$

This is a definition and a theorem at the same time, because we need to prove that if one of the three conditions holds, then α indeed always preserves the rule r. We start with an informal consideration of each of the three cases. In the first case, α does not negate any literal of r, and thus if the rule r holds before the execution of α, it still holds after the execution. In the second case, since α establishes some literal of r, this rule always holds after the execution of α. The third case describes the situation when the execution of α may lead to a state in which the rule r does not hold. This happens if none of the literals $l_{k+1}, l_{k+2}, \ldots, l_n$ holds before the execution of α, and α negates the rest of literals of the rule r during its execution. However, the rules $r_1, r_2, \ldots, r_k$ guarantee that in such a situation the rule r does not hold before the execution of α, and therefore the rule is not violated by the execution. Thus, in all three cases, if r holds before the execution of α, it still holds after the execution.

Theorem 2.1 *Consider a problem domain with some invariant I. The rule r is an invariant rule if and only if it is preserved by every operator in the problem domain.*

Proof. Assume that r is preserved by every operator. We need to show that for every state S satisfying r, and for every α legal in S, $\alpha(S)$ satisfies r. If $k = 0$, then no literal of r is changed by α, and therefore r holds in $\alpha(S)$. If there exists $l \in r$ such that $l \in \mathit{Eff}(\alpha)$,

then $l \in \alpha(S)$, and therefore $\alpha(S)$ satisfies r. Finally, if $r_1, \ldots, r_k$ are invariant rules, then S satisfies all these rules. Since S satisfies r, one of the literals $l_1, \ldots, l_k$, say l_i, holds in S, and, since $Pre(\alpha) \subseteq S$, none of the literals $\neg l'_1, \ldots, \neg l'_m$ holds in S. Thus, from the rule r_i we conclude that one of the literals $l_{k+1}, \ldots, l_n$, say l_j, holds in S. Since $\neg l_j \notin Eff(\alpha)$, it still holds in $\alpha(S)$, and therefore in this case $\alpha(S)$ also satisfies r.

Now assume that there exists some operator α that does not preserve r, that is all three conditions stated before the theorem do not hold. Then $k > 0$, that is α negates al least one literal of the rule r, and one of the rules $r_1, \ldots, r_k$ for the operator α, say r_i, is not an invariant rule, and therefore there exists a state S that does not satisfy r_i. Then all of the literals l_i, $\neg l_{k+1}, \ldots, \neg l_m$, and $l'_1, \ldots, l'_m$ hold in S. Therefore, S satisfies r (due to the literal l_i) and $Eff(\alpha) \subseteq S$. Since α establishes $\neg l_1, \ldots, \neg l_k$, and does not establish any of the literals $l_{k+1}, \ldots, l_n$, we conclude that $\neg l_1, \ldots, \neg l_n \in \alpha(S)$, and therefore $\alpha(S)$ does not satisfy r. Thus, r may be violated by an application of α and therefore r is not an invariant rule. □

We did not find an efficient way to use this theorem for generating invariants for a problem domain. However, it may be used to check whether some given set of rules I is an invariant.

2.5 Criticalities

The *criticality* of a variable is some number (usually natural) that characterizes the "importance" of the variable in the planning domain. While achieving some goal, we first try to achieve values of "important" variables. Later we may refine our plan by adding operators to achieve values of less important variables.

An abstract problem space contains only variables with criticalities not less than some fixed number. While planning in an *abstract space*, we ignore all variables with smaller criticalities by removing them from the preconditions of the operators and from the goal. The set of the criticalities of variables in a problem domain is called an *abstraction hierarchy*.

Abstract planning is usually done in a top-down manner. First we find a plan that solves the goal at the highest *level of abstraction*, that is we "pay attention" only to the variables with the highest criticality. Then we refine this plan at successively lower levels by inserting new operators to achieve the reintroduced preconditions which were ignored during planning at the higher levels. When we refine a plan at some level of abstraction, we usually try to preserve the values of variables at higher levels. This guarantees that we cannot accidentally violate the correctness of the higher-level plan. However, the requirement to preserve the abstract-level values is *not* strict. In Chapter 4 we describe a planning algorithm that changes the values of higher-level variables while planning at a concrete level, but then modifies the resulting plan in such a way that the final version of the concrete-level plan still preserves the structure of the initial abstract-level plan.

Formally, an *abstraction hierarchy* H is a pair $(\mathcal{X}, crit)$, where $\mathcal{X}$ is the set of variables in a problem domain, and $crit$ is a function from $\mathcal{X}$ into the set of natural numbers. A *level* of an abstraction hierarchy is formed by the domain variables with the same criticality. For two hierarchies, $H_1 = (\mathcal{X}, crit_1)$ and $H_2 = (\mathcal{X}, crit_2)$, in the same problem domain, we say

that H_1 is *finer-grained* than H_2 if, for any two domain variables $x_1, x_2 \in \mathcal{X}$,

- $crit_1(x_1) \leq crit_1(x_2)$ if and only if $crit_2(x_1) \leq crit_2(x_2)$, and
- if $crit_1(x_1) = crit_1(x_2)$ then $crit_2(x_1) = crit_2(x_2)$

Intuitively, H_1 is either the same as H_2 or obtained from H_2 by dividing some of the levels of H_2 into smaller sublevels.

Let α be an operator, and i be some criticality value. We denote the set of preconditions of α that have criticality values more than or equal to i by ${}_iPre(\alpha)$:

$$\begin{aligned} {}_iPre(\alpha) &= \{(x = v) \mid (x = v) \in Pre(\alpha) \text{ and } crit(x) \geq i\} \\ &\quad \cup \{(x \neq v) \mid (x \neq v) \in Pre(\alpha) \text{ and } crit(x) \geq i\} \end{aligned}$$

We denote the operator with preconditions ${}_iPre(\alpha)$ and effects $Eff(\alpha)$ by ${}_i\alpha$ and the set of all such ${}_i\alpha$ by ${}_iO$, that is

$${}_iO = \{{}_i\alpha \mid \alpha \in O\}$$

The problem space at level i of abstraction is defined by the set of variables $\mathcal{X}$ and the set of operators ${}_iO$.

Similarly, we can abstract the goal state:

$${}_iS_g = \{(x = v) \mid (x = v) \in S_g \text{ and } crit(x) \geq i\}$$

The *Upward Solution Property* proved in [Tenenberg, 1988] states that if a plan is correct at some level of a hierarchy, it is also correct at any higher level.

Example

We consider the tower of Hanoi problem described in the previous example. Intuitively, the "most important" variable is the position of the large disk, *Where_L*, because once the large disk is put onto the proper peg, the other two disks may be put into their positions without moving the large one. A similar reasoning shows that *Where_M* is the second important variable, and *Where_S* is the least important. So, we assign criticalities such that

$$crit(Where_S) < crit(Where_M) < crit(Where_L)$$

Thus, the criticality assignment is as follows:

$$\begin{aligned} crit(Where_L) &= 2 \\ crit(Where_M) &= 1 \\ crit(Where_S) &= 0 \end{aligned}$$

Recall that our initial state is

$$S_g = \{(Where_S = 1), (Where_M = 2), (Where_L = 2)\}$$

On the highest level of abstraction we ignore all variables except *Where_L*, and the goal becomes

$$S_g = \{(Where_L = 2)\}$$

This goal may be achieved by a single operator:

$$\rightarrow Move_L(1,2) \rightarrow$$

The only precondition of this operator on the highest level of abstraction is $(Where_L = 1)$. On the next level of abstraction we take into account the variable *Where_M*, but still ignore *Where_S*. On this level of abstraction the goal is

$$S_g = \{(Where_M = 2), (Where_L = 2)\}$$

We need to refine our one-operator plan by adding some operators before $Move_L(1,2)$ so that the preconditions $(Where_M \neq 1)$ and $(Where_M \neq 2)$ of $Move_L(1,2)$ become true before its execution, and we need to add some operators after it to make $(Where_M = 2)$ hold in the final state. After adding such operators, we receive the following plan:

$$\rightarrow Move_M(1,4) \rightarrow Move_L(1,2) \rightarrow Move_M(4,3) \rightarrow$$

Finally, we consider the lowest level of abstraction, where all three variables in the problem domain matter. At the lowest level the above plan is incorrect: its first operator, $Move_M(1,4)$, cannot be applied in the initial state, because its precondition $(Where_S \neq 1)$ is not satisfied. So, we need to add some new operators to make the plan correct. Recall, that we cannot change or reorder operators of the higher-level plan, and we cannot add operators that change values of higher level variables. Thus, we may use only the operator *Move_S*. One of the possible refinements of the above plan is

$$\rightarrow Move_S(1,3) \rightarrow Move_M(1,4) \rightarrow Move_L(1,2) \rightarrow Move_M(4,2) \rightarrow Move_S(3,1) \rightarrow$$

It is easy to check that this plan is correct and achieves the goal. □

When we deal with hierarchies that have some "good" property (for example, the *ordered property* [Knoblock *et al.*, 1991] or the *downward refinement property* [Bacchus and Yang, 1991]), efficiency of planning usually (but not always) increases as the number of abstraction levels increases. Thus, we wish to have as many levels of abstraction as possible as long as "good" properties are preserved. However, the evidence for high efficiency of planning in finer-grained hierarchies is mostly empirical. Some theoretical work has been done to demonstrate high efficiency of planning in multilevel ordered hierarchies [Knoblock, 1991b], [Bacchus and Yang 1992], but the results presented in these papers are based on assumptions that are too strong for most planning problems.

2.6 Data structures and basic algorithms

All sets and relations discussed above must somehow be represented in the computer memory. (We use the real RAM model of computation.) The reader who is not interested in the details of implementation may skip this section.

We represent sets as arrays or linked lists. Such a representation allows us to retrieve consecutively all elements of a set in linear time. We keep the set of variables $\mathcal{X}$ as two disjoint

```
Check_Preconditions_1(S, α)
1. for each (x = v) ∈ Pre(α) do
2.     if S(x) ≠ v
3.         then /* preconditions are not satisfied */ return(False)
4. /* preconditions are satisfied */ return(True)

Apply_1(S, α)
1. for each (x = v) ∈ Eff(α) do
2.     S(x) := v;
3. return(S)
```

Table 2.4: Simple algorithms for the full representation

sets, the set of changeable variables $\mathcal{X}_c$ and the set of unchangeable variables $\mathcal{X}_u = \mathcal{X} - \mathcal{X}_c$. For every variable x, we keep two disjoint sets of its values: the set of achievable values $D_a(x)$, and the set of unachievable values $D_u(x)$. For each operator we keep pointers to all its preconditions and effects so that we can quickly retrieve them. Observe that these pointers allow us to retrieve in linear time not only the preconditions and the effects, but also the outcomes of an operator. For each variable x we keep the set of pointers to every operator that changes x (that is to every operator of the set O_x), and for each value v of x we keep the set of pointers to every operator that achieves $(x = v)$ (that is to every operator of the set $O_{(x=v)}$).

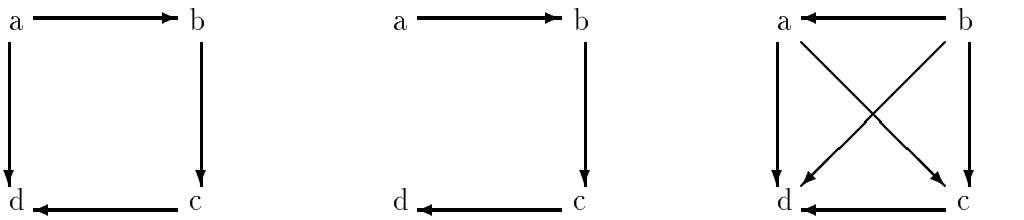
There are two ways to keep the description of the current state S of the world. The first method is to keep S as an array indexed on the variables of $\mathcal{X}$, that is S has $|\mathcal{X}|$ entries, each corresponding to a particular variable. Each planning variable x may accept either some value from its domain $D(x)$ or the special value *Unknown*, indicating that the value of x is not specified in the state S. We call this representation the *full representation* of S. The advantage of this representation is that the value of each variable may be accessed in constant time. We may check whether the preconditions $Pre(\alpha)$ of some operator α hold in the state S in $O(|Pre(\alpha)|)$ running time. Table 2.4 present the algorithm *Check_Preconditions_1*, which checks whether the preconditions of α hold in S. (Recall that the notation "$S(x)$" stands for the value of x in the state S). The comments in the algorithm are bracketed by $/* \ldots */$. Also, we may find the result of applying α to S, that is the new state $\alpha(S)$, in $O(|Eff(\alpha)|)$ time. We find $\alpha(S)$ by assigning a new value to each variable changed by α. The algorithm *Apply_1* that finds $\alpha(S)$ is presented in Table 2.4.

The full representation of S is convenient if $\mathcal{X}$ is small, or if the values of most variables are specified (not *Unknown*) in S. If $\mathcal{X}$ is large and only few variables have known values, the representation described above becomes inefficient because it takes to much memory. It is more efficient to keep only the variables whose values are specified in S. In this case we assume that the value of a variable is specified in the state S *if and only if* it is indicated in the representation of S. This assumption is known as the *Closed World Assumption* [Genesereth and Nilsson, 1987]. We call this representation the *closed-world representation* of S. To be able to access elements of S quickly, we represent S as a *black-red tree* indexed on the names of variables. A black-red tree is a kind of a binary-search tree. Search of an

Check_Preconditions_2(S, α)
1. **for** each $(x = v) \in Pre(\alpha)$ **do**
2. /* find the node with the value of x */ Find(S, x);
3. **if** (node with x not found) **or** $(S(x) \neq v)$
4. **then** /* preconditions are not satisfied */ return(*False*)
5. /* preconditions are satisfied */ return(*True*)

Apply_2(S, α_1)
1. **for** each $(x = v) \in Eff(\alpha)$ **do**
2. /* find the node with the value of x */ Find(S, x);
3. **if** node with x not found in S
3. **then** /* insert a new node */ Insert$(S,(x = v))$
4. **else** /* change the value of x */ $S(x) := v$;
5. return(S)

Table 2.5: Simple algorithms for the closed-world representation

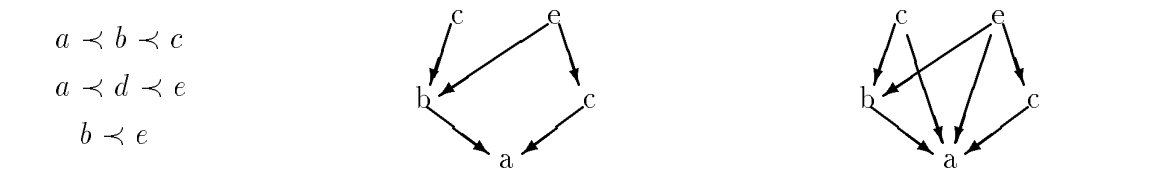


(a) Graph (b) Transitive reduction (c) Transitive closure

Figure 2.5: Example of the transitive reduction and transitive closure

element in a black-red tree and insertion of a new element take $O(\log n)$ time in the worst case, where n is the number of nodes in the tree. The description of a black-red tree and algorithms on it may be found in [Cormen *et al.*, 1990]. Thus, in this case the access to the value of a variable in the description of the state S takes $O(\log |S|)$ time. The algorithm for checking whether the preconditions of an operator α are satisfied in a state S in this case takes $O(|Pre(\alpha)| \cdot \log |S|)$ time, and the algorithm for computing $\alpha(S)$, which is the result of applying an operator α to a state S, takes $O(|Eff(\alpha)| \cdot \log |S|)$ time. The both algorithms are presented in Table 2.5.

We keep partially ordered sets as *acyclic directed graphs*. A *directed graph* is a set of *vertices* some of which are connected with *directed edges*, each directed edge pointing from one vertex to another. An example of a directed graph is shown in Figure 2.5a. We denote a directed edge from a vertex a to a vertex b by (a, b). A *path* in a directed graph is a sequence of vertices $(a_1, a_2, \ldots, a_n)$ such that there is the edge from a_1 to a_2, from a_2 to a_3, and so on. For example, the graph in Figure 2.5a contains a path (a, b, c, d). A graph is called *acyclic* if it does not contain any path of the form $(a_1, a_2, \ldots, a_n, a_1)$. Intuitively, if we travel along the edges in an acyclic graph, then, once we have left some vertex, we can never visit it again.



(a) Partial order (b) Representation by a transitively reduced graph (c) Representation by a transitively closed graph

Figure 2.6: Graph representation of a partial order

Two important kinds of directed graphs are *transitively reduced* and *transitively closed* graphs.

Definition 2.1 (Transitive edges)
Let G be a directed graph, and a and b be its vertices. The edge (a, b) from a to b is called transitive *if there is a path from a to b that does not contain the edge (a, b).*

Definition 2.2 (Transitively reduced and transitively closed graphs)
An acyclic directed graph is called transitively reduced *if it does not contain any transitive edges. A graph is* transitively closed *if it contains all possible transitive edges, that is no transitive edge may be added to the graph.*

Figure 2.5 presents an example of the transitive reduction and transitive closure of an acyclic graph. Each acyclic graph has exactly one transitive closure and transitive reduction [Cormen *et al.*, 1990].

We use an *adjacency list* to represent a directed graph in the computer memory [Cormen *et al.*, 1990]. This means that for each vertex a of the graph we keep two linked list. One of the lists contains all such vertices b that there is an edge from a to b. The other list contains all such vertices c that there is an edge from c to a.

Since the time-precedence relation on the operators of a nonlinear plan is a partial order, we need a data structure to represent partially ordered sets. We represent a partially ordered set in computer memory as two directed graphs: a transitively reduced and transitively closed graph. To construct a transitively closed graph for some partially ordered set, we represent each element of the set as a vertex of the graph, and for each two elements a and b such that $a \prec b$, we draw an edge from b to a. The resulting graph is acyclic because the partial order is antisymmetric, and the graph is transitively closed because the partial order is transitive [Enderton, 1977]. An example of the graph representation of a partial order is shown in Figure 2.6.

The representation of a partially ordered set as a transitively closed graph, stored in the computer memory as an adjacency matrix, allows us to compare two elements of the set, that is two find which of the two elements is larger under the partial order, in constant time. To compare a and b, we just check if there is an edge in the graph connecting a and b. If this edge goes from b to a, then $a \prec b$. If there is no edge between a and b, they are

incomparable. We use this method to determine, in constant time, which of two operators of a nonlinear plan is executed earlier.

Also, we may remove an element from a transitively closed graph, preserving the transitively closed structure of the remaining graph, in $O(V)$ time, where V is the number of vertices in the graph. To do this, we just remove the element, all edges incoming into this element, and all edges outcoming from it.

To construct a transitively reduced graph, we remove all transitive edges from the transitively closed graph described above. Observe that if there is a path from a vertex a to a vertex b in a graph, then a path from a to b is also exists in the transitive reduction of the graph [Cormen *et al.*, 1990]. Thus, for transitively reduced representation of a partially ordered set we have: $a \prec b$ if and only if there is a path from b to a.

For a given element a in a partially ordered set A, we define the set of immediate successors of a by

$$Immediate_Successors(a) = \{b \in A \mid b \prec a \text{ and } (\nexists c \in A)\ b \prec c \prec a\}$$

and the set of immediate predecessors of a by

$$Immediate_Predecessors(a) = \{b \in A \mid b \succ a \text{ and } (\nexists c \in A)\ b \succ c \succ a\}$$

The vertices in the transitively reduced graph that have outcoming edges leading to a correspond to the set of immediate predecessors of a. The edges outcoming from a in the transitively reduced graph point to the immediate successors of a. For example, the immediate successor of b in Figure 2.6b is a, and the immediate predecessors are c and e. So, the transitively reduced graph allows us to find the set of immediate successors and predecessors of a given element in linear time.

To keep the order of operators of a nonlinear plan Π in the computer memory, we represent Π as two directed graphs: a transitively reduced graph Π_r and a transitively closed graph Π_c. Vertices of these graphs correspond to operators of the plan Π.

To work effectively with graphs, we must be able to perform several simple operations on them.

1. Linearization

Given a set $\{a_1, a_2, \ldots, a_n\}$, and a partial order $\prec$ on this set, we wish to find a sequence $(a_{k_1}, a_{k_2}, \ldots, a_{k_n})$ of elements of the set where for each pair of comparable elements, the smaller element occurs before the larger one. Formally, for all $i, j \in [1..n]$, if $i < j$ then either $a_{k_i} \prec a_{k_j}$, or a_{k_i} is incomparable with a_{k_j}.

If the partial order is represented as a directed graph, this problem is called the *topological sorting*. The running time of the algorithm that performs the topological sorting is $O(V+E)$, where V is the number of vertices in the graph, and E is the number of edges. The algorithm is described in [Aho *et al.*, 1974].

2. Combining strongly connected components

A *strongly connected component* of a directed graph is a maximal set of vertices such that there is a path from each vertex of the set to each other vertex of the set (see Figure 2.7). We wish to replace each strongly connected component with a single vertex, whose incoming

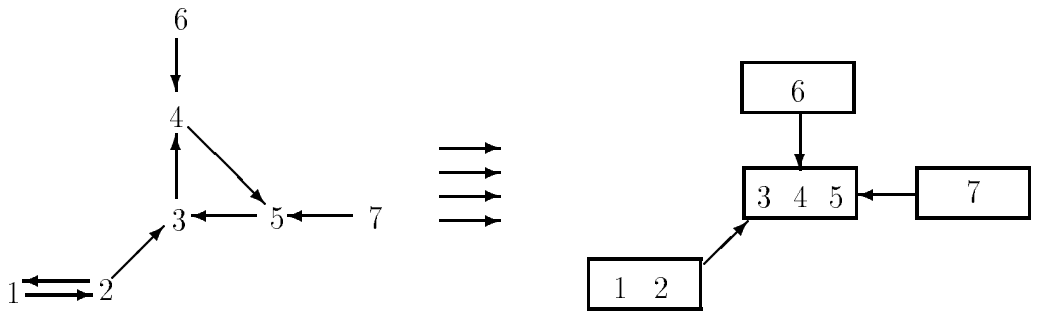


Figure 2.7: Example of combining strongly connected components

edges are all the edges that go from outside into the component, and whose outcoming edges are all the edges that come out of the component. Formally, if $\{a_1, a_2, \ldots, a_n\}$ is the set of vertices in a strongly connected component in a graph G, we remove all these vertices with all their incoming and outcoming edges, and insert a single vertex a with the set of outcoming vertices

$$\{(a, b) \mid (\exists i \in [1..n])\ (a_i, b) \text{ was in } G\}$$

and the set of incoming edges

$$\{(b, a) \mid (\exists i \in [1..n])\ (b, a_i) \text{ was in } G\}$$

A solution of this problem with running time $O(V^2)$ is presented in [Cormen *et al.*, 1990].

3. Finding the set of minimal elements

A *minimal element* a of a partially ordered set A is such an element that

$$(\nexists b \in A)\ b \prec a$$

that is all other elements of A are either larger than a or incomparable with a. Let a partial order on A be represented by an acyclic directed graph G. We wish to find the set Y of minimal elements of A. (We do *not* assume that G is transitively reduced or transitively closed.)

The algorithm that finds the set of minimal elements is presented in Table 2.6. To understand how it works, observe that if there is an edge from a vertex a to some other vertex b, then $b \prec a$, and therefore a is *not* minimal. On the other hand, if there are no edges outcoming from a, then there are no elements less than a, and a *is* a minimal element. The adjacency-list representation of a graph allows us to check the condition in line 3 of the algorithm in constant time. To do this, we just check whether the list of the edges outcoming from a is empty. The total running time of the algorithm is $O(V)$, where V is the number of vertices in the graph.

4. Transitive reduction

Given an acyclic directed graph, we wish to find its transitive reduction, that is to delete all its

```
Minimal_Elements(G)
1. Y := ∅;
2. for every vertex a of G do
3.     if there are no outcoming edges from a
4.         then Y := Y ∪ {a};
5. return(Y)
```

Table 2.6: Finding minimal elements of partially ordered set represented by the graph G

transitive edges. An efficient transitive-reduction algorithm is presented in [K. Simon, 1985]. Its worst-case time is $O(V \cdot E_{red})$, where V is the number of vertices in the graph, and E_{red} is the number of edges *after* the transitive reduction. The average-case running time of the algorithm is $O(V^2 \cdot \log\log V)$.

5. Transitive closure

Given an acyclic directed graph, we wish to find its transitive closure, that is to add all possible transitive edges to the graph. For every two vertices a and b such that there is a path from a to b, we add an edge from a to b.

The algorithm that solves this problem is presented in [K. Simon, 1985]. The running time of the algorithm is the same as the running time of the transitive reduction algorithm: the average-case running time is $O(V^2 \cdot \log\log V)$, and the worst-case running time is $O(V \cdot E_{red})$, where V is the number of vertices in the graph, and E_{red} is the number of edges in the transitively reduced version of the graph.

Chapter 3

Justified plans

When we search for a plan to achieve certain goal, we wish to find a plan that does not contain "useless" steps. In other words, we wish to optimize the plan by removing all operators that are not necessary for achieving the goal. For example, suppose one has a kettle with water and wishes to obtain a cup of a hot water, by following the plan:

1. Boil water in the kettle.
2. Pour water into the cup.

If later one discovers that the kettle already contains hot water, then the first step of the plan, *"boil water"*, is no longer necessary for achieving the goal. After removing the first step, the resulting plan,

1. Pour water into the cup.

contains fewer steps while still achieving the same goal. The operation of removing useless operators from a plan is known as *justification*. The main purpose of this chapter is to formalize different ways of performing justification.

One application of the justification would be to augment a non-optimal planner such as STRIPS with a justification routine. The resulting plan will then be more efficient to execute. This may be especially useful for planners that use macro operators (such as the planner described in [Korf, 1985]), because the plans after substituting atomic operators for macro operators are often non-optimal. Hierarchical planners and planners that use primary effects of operators also may produce non-optimal plans, because the optimal plans may be ruled out by restrictions imposed by the planning process. Finally, even planning algorithms based on the A^* search technique (e.g. TWEAK) may produce non-optimal plans if the heuristic used by A^* does not guarantee optimality.

Another application of this optimization is in reusing old plans. Suppose that we have found a plan for achieving goals G_1, G_2, and G_3. Later we may use the same plan to achieve the goal G_1 alone. In this case we wish to find the subplan of the initial plan which is "relevant" to achieving G_1, by removing all unnecessary operators. Thus, justification would be useful for adapting old plans to new situations.

The notion of justified plans is important not only for the purpose of optimizing plans, but also for abstract problem solving. Several important concepts describing the algorithms

for generating abstraction hierarchies are defined via justified plans. For example, the theoretical concepts underlying Knoblock's planner ALPINE [Knoblock, 1990] are based on the notions of justified plans. Other results that depend on this notion are presented in [Yang and Tenenberg, 1990], [Knoblock *et al.*, 1991], and [Bacchus and Yang, 1991].

In spite of the importance of the concept of justified plans, relatively few efforts have been made to explore different kinds of justification. In this chapter we begin to fill this gap by formalizing, unifying, and extending the previous work. First we consider the notion of *backward justified* plans, which guarantees that each operator in the plan establishes a literal necessary for achieving the goal. Then we present a syntactic definition of *well-justified* plans. Informally, a plan is well-justified if none of its operators may be omitted. We then compare well-justified and backward justified plans. Finally, we consider the task to find the "best possible" justification of a given plan, that is a subplan of a given plan that cannot be further optimized by removing any subset of its operators. We show that the task of finding such a subplan is NP-complete. To satisfy the practical need for efficient planning, we present a greedy algorithm that finds a near-perfect justification in polynomial time.

In the next chapter we will show that the notions of different kinds of justification allow us to introduce different kinds of ordered hierarchies. This will lead to increasing the efficiency of Knoblock's planner ALPINE by generating finer-grained ordered hierarchies than those generated by ALPINE.

3.1 Backward justification

To formalize the notion of justified plans, we first generalize the concept of establishment relation defined in [Knoblock *et al.*, 1991] to nonlinear plans.

Definition 3.1 (Establishment) *Let* $(S_0, S_g, \overline{\Pi})$ *be a correct linear plan. Let* α_1 *and* α_2 *be two operators of the plan,* $\alpha_1, \alpha_2 \in \overline{\Pi}$, $(x = v) \in \mathrm{Eff}(\alpha_1)$, *and* $(x = v) \in \mathrm{Pre}(\alpha_2)$. *Then* α_1 establishes $(x = v)$ for α_2 *if*

- $\alpha_1 \prec \alpha_2$, *and*
- $\forall \alpha \in \overline{\Pi}$, *if* $\alpha_1 \prec \alpha \prec \alpha_2$ *then* $x \not\sqsubset \mathrm{Eff}(\alpha)$

We say that α_1 possibly establishes *a value* $(x = v)$ *for* α_2 *in a nonlinear plan* (S_0, S_g, Π) *if it establishes* $(x = v)$ *for* α_2 *in at least one linearization of* (S_0, S_g, Π).

Intuitively this means that the precondition $(x = v)$ of the operator α_2 holds before the execution of α_2, and α_1 is the last operator that achieves it.

Definition 3.2 (Backward justification) *Let* (S_0, S_g, Π) *be a correct plan. An operator* $\alpha \in \overline{\Pi}$ *is called* backward justified *if* $\exists (x = v) \in \mathrm{Eff}(\alpha)$ *such that* α *possibly establishes* $(x = v)$ *either for the goal* S_g *or for another backward justified operator.*

We say that a plan Π is backward justified, if all its operators are backward justified. This definition of justification was used in the planner ALPINE [Knoblock *et al.*, 1991]. For linear plans it is equivalent to the definition stated in [Yang and Tenenberg, 1990]. For nonlinear plans, backward justification is weaker than the justification described in [Yang and Tenenberg, 1990].

Intuitively, an operator is backward justified if it establishes some literal necessary for achieving the goal. However, it may happen that this literal has been established before α, and then α is useless for our plan. So, backward justified operators are not "truly justified". We illustrate this point with the following example.

Assume you have a kettle with a hot water and an empty cup, and you wish to have a cup of hot water. The following plan achieves the goal

1. Pour water into the cup.
2. Heat the water in the cup by putting the cup into a microwave.

The second operator *is* backward-justified, because it makes the water hot, while no other operator *after* it achieves the same goal of making the water hot in the final state. However, this operator still may be skipped, because the water was already hot before its execution. Thus, the second operator is not "truly justified".

The following theorem was stated in [Yang and Tenenberg, 1990] for the definition of justified operators used in ABTWEAK. We show that it also holds for our definition of backward justified plans.

Theorem 3.1 *Let (S_0, S_g, Π) be a correct plan. Its correct backward justified subplan may be found by removing all non-backward-justified operators.*

Proof. Let Π' be the plan obtained from Π by removing all non-backward justified operators. We need to show that this new plan is correct and backward justified.

Claim 1. (S_0, S_g, Π') is correct.

We need to show that every linearization of Π' is correct. So consider its arbitrary linearization $\overline{\Pi}'$, and let $\overline{\Pi}$ be such a linearization of Π that $\overline{\Pi}'$ is a subplan of $\overline{\Pi}$. (Such a linearization of Π exists by Lemma 2.2). $\overline{\Pi}$ is correct as a linearization of a correct plan. Now consider an arbitrary operator α of $\overline{\Pi}'$. In $\overline{\Pi}$, all preconditions of α are satisfied. Since $\overline{\Pi}'$ is obtained from $\overline{\Pi}$ by removal of non-backward-justified operators, none of removed operators may establish any precondition of α. Therefore, all preconditions of α are established either by the initial state or by backward justified operators, and therefore these preconditions still hold in $\overline{\Pi}'$ before α. Therefore, α is legal in $\overline{\Pi}'$. Thus, we have shown that all operators of $\overline{\Pi}'$ are legal. The proof that $\overline{\Pi}'$ achieves the goal is similar.

Claim 2. (S_0, S_g, Π') is backward justified.

Let α be an arbitrary operator of Π'. We need to show that α is backward justified in at least one linearization of Π'. Since α is backward justified in Π, it must be justified in one of the linearizations of Π. Let us denote this linearization by $\overline{\Pi}$. We remove from $\overline{\Pi}$ all operators that are not backward justified in Π and obtain a new linear plan $\overline{\Pi}'$. Observe that according to Claim 1, Π' is correct, and therefore $\overline{\Pi}'$ is also correct, as a linearization of a correct plan.

The operator α is backward justified in $\overline{\Pi}$, and we wish to show that it is backward justified in $\overline{\Pi}'$. For this purpose we show that *every backward justified operator of* $\overline{\Pi}$ *is also backward justified in the plan* $\overline{\Pi}'$. We denote operators of $\overline{\Pi}'$ by α_1,α_2,...,α_k. To prove our

claim, we use induction in backward direction: we show that if our hypothesis holds for $\alpha_{i+1}, \ldots, \alpha_k$, then it holds for α_i as well.

Base: $i = k + 1$, that is the induction hypothesis holds for the empty set of operators. This is a trivial case.

Step: Now assume that the hypothesis holds for $\alpha_{i+1}, \ldots, \alpha_k$. We need to show that if α_i is backward justified in $\overline{\Pi}$, it is also backward justified in $\overline{\Pi}'$. So assume α_i is justified in $\overline{\Pi}$.

Case 1. α_i establishes some value $(x = v)$ for the goal. Then, by the definition of establishment, no operator in $\overline{\Pi}$ after α_i changes x. Since $\overline{\Pi}'$ is a subplan of $\overline{\Pi}$, no operator in $\overline{\Pi}'$ after α_i changes x either, and therefore α_i establishes $(x = v)$ for the goal in the plan $\overline{\Pi}'$. Therefore, α_i is backward justified in $\overline{\Pi}'$.

Case 2. α_i establishes a value $(x = v)$ for some backward justified operator α' of $\overline{\Pi}$. Observe that since α' is backward justified in $\overline{\Pi}$, it is also backward justified in Π, and therefore, by construction of $\overline{\Pi}'$, α' is one of the operators $\alpha_{i+1}, \ldots, \alpha_k$. By the definition of establishment, no operator of $\overline{\Pi}$ between α_i and α' changes x. Therefore, no operator of $\overline{\Pi}'$ between α_i and α' changes x either, and thus α_i establishes $(x = v)$ for α' in the plan $\overline{\Pi}'$. By inductive hypothesis, α' is backward justified in $\overline{\Pi}'$. Therefore, α_i is also backward justified in $\overline{\Pi}'$. □

The algorithm for removing non-backward-justified operators is shown in Table 3.1. The algorithm begins by considering the last operator of the plan. If this operator does not contribute any value into achieving the goal, it is removed. Then the algorithm considers the second last operator, then the third last, and so on till the first operator of the plan. If the plan is nonlinear, we may use the order defined by any of its linearizations. Each operator that does not establish a value of any variable for the goal nor for any other operator is removed. Observe that when we consider an operator, all non-backward-justified operators after this operator are already removed. Thus, the operator is not removed only if it establishes a precondition for some *backward justified* operator, which means that the operator itself is backward justified. (Notice that the algorithm proceeds from the end to the beginning of the plan. This is the reason for the term "*backward* justified").

The algorithm *Possibly_Establish*(α,α_1,$(x = v)$) checks whether α possibly establishes $(x = v)$ for α_1. It checks every operator α_2 which changes x, and if α_2 is necessarily between α and α_1, it concludes that α does *not* establish $(x = v)$ for α_1. If the order of operators is represented as a transitively closed graph, the conditions in line 2a may be checked in constant time, and therefore the algorithm *Possibly_Establish* runs in $O(|\Pi|)$ time.

Lines 5–7 of the algorithm *Backward_Justified* are executed once for each effect of each operator, and so the total number of executions is $E = \sum_{\alpha \in O} |\mathit{Eff}(\alpha)|$. Within lines 5–7 the procedure *Possibly_Establish* is called $|\Pi|$ times, and the procedure itself takes $O(|\Pi|)$ time. So the total running time of the algorithm *Backward_Justified* is $O(E \cdot |\Pi|^2)$.

The main advantage of a backward justification is that it can be found *quickly*. The algorithm described above works faster than the justification algorithm used in ABTWEAK

```
Backward_Justification(S_0, S_g, Π)
1.  let Π̄ be some linearization of Π;
2.  for α := (last operator of Π̄) downto (first operator of Π̄) do
    begin
3.      Justified := False;
4.      for each (x = v) ∈ Eff(α) do
        begin
5.          for each α_1 in Π such that (x = v) ∈ Pre(α_1) do
6.              if Possibly_Establish(α,α_1,(x = v))
7.                  then /* α is backward justified */ Justified := True;
8.          if Possibly_Establish(α, S_g, (x = v))
9.              then /* α is backward justified */ Justified := True
        end;
10.     if Justified=False /* α is not backward justified */
11.         then remove α from the plan Π
    end

Possibly_Establish(α,α_1,(x = v))
1a. for every α_2 ∈ Π such that x ⊏ Eff(α_2) do
2a.     if α ≺ α_2 and α_2 ≺ α_1
3a.         then /* α does not establish (x = v) for α_1 */ return(False);
4a. return(True)
```

Table 3.1: Finding the backward justified subplan of a given plan

[Yang and Tenenberg, 1990][1], and much faster than other justification algorithms that we discuss in this thesis.

3.2 Well-justification

We have shown in the previous section that the plan

Initial state: A kettle of hot water.
1. Pour water into the cup.
2. Heat the water in the cup by putting the cup into a microwave.

Goal state: A cup of hot water.

is backward-justified, while its second operator still may be skipped. In this section we introduce a stronger justification, called a *well-justification*, that does not contain any operator that may be skipped without violating the correctness of the plan.

Definition 3.3 (Well-justification) *An operator α_i in a linear plan $(S_0, S_g, \overline{\Pi})$ is called* well-justified *if $\exists (x = v) \in \mathrm{Eff}(\alpha_i)$ such that α_i establishes $(x = v)$ for some operator or for the goal S_g, and $(x = v)$ does not hold before the execution of α, that is $(x = v) \notin S_{i-1}$. An operator in a nonlinear plan is called well-justified if it is well-justified in at least one linearization of the plan.*

We say that a plan is well-justified if all its operators are well-justified. Intuitively, an operator is well-justified if it establishes a value of some variable which has not been established before, and which is necessary for executing some other operator. This means that if we remove a well-justified operator from a plan, the plan is no longer correct. We state it as a lemma.

Lemma 3.1 *An operator is well-justified if and only if we cannot remove it from the plan without violating the correctness of the plan.*

Proof. Assume α is a well-justified operator of a plan (S_0, S_g, Π). To prove that α cannot be removed from (S_0, S_g, Π) without violating its correctness, we need to show that α cannot be removed from at least one linearization of (S_0, S_g, Π). Let $(S_0, S_g, \overline{\Pi})$ be a linearization of (S_0, S_g, Π) in which α is well-justified. Then α establishes some value $(x = v)$ for some operator of $\overline{\Pi}$ or for S_g. Let us consider the former case. By the definition of establishment, no operator between α and α_1 changes x, and by the definition of well-justification, the value of x before α is different from v. Thus, if we remove α, the value of x before α_1 is different from $(x = v)$, and the plan is no longer correct. The situation when α establishes $(x = v)$ for the goal is treated similarly.

Now assume that α is *not* well-justified. We need to show that it may be removed from any linearization of (S_0, S_g, Π) without violating the correctness of the linearization. Consider an arbitrary linearization $(S_0, S_g, \overline{\Pi})$. If α establishes a value $(x = v)$ for some operator

[1] ABTWEAK justification takes $O(P \cdot E \cdot |\Pi|)$ time, where $P = \sum_{\alpha \in O} Pre(\alpha)$ (Yang, private communications).

α_1 (or for the goal), then $(x = v)$ holds before α, and no operator between α and α_1 changes x. Therefore, $(x = v)$ will hold before α_1 after the removal of α. Thus, all preconditions of operators established by α still hold after removing α, and therefore the removal of α does not violate the correctness of the plan. □

The next theorem follows directly from the lemma.

Theorem 3.2 *A plan is well-justified if and only if there is no operator that can be removed without violating the correctness of the plan.*

This theorem shows that well-justification captures the intuition behind "good" plans: a well-justified plan does not contain any operator that is not necessary for achieving the goal. The next theorem shows that well-justification is stronger than backward justification.

Theorem 3.3 *If a plan is well-justified, it is also backward justified.*

Proof. Consider some plan (S_0, S_g, Π), and assume that this plan is not backward justified. We need to show that it is not well-justified either. Let α be a non-backward-justified operator of Π such that no non-backward-justified operator is necessarily after α. In other words, there is no such a non-backward-justified operator α_1 that $\alpha \prec \alpha_1$. To prove that (S_0, S_g, Π) is not well-justified, we show that the operator α is not well-justified. So we need to show that α is not well-justified in every linearization of Π.

Let $\overline{\Pi}$ be an arbitrary linearization of Π. Since α is not backward justified, it does not establish any value for the goal, nor for any backward justified operator. If α' is a non-backward-justified operator after α, α *may* establish some precondition $(x = v)$ of α'. However, we will show that in this case $(x = v)$ holds before α. Let α be the k-th operator of $\overline{\Pi}$, and α' be the m-th operator:

$$\overline{\Pi} = (\alpha_1, \alpha_2, \ldots, \alpha_{k-1}, \alpha, \alpha_{k+1}, \ldots, \alpha_{m-1}, \alpha', \alpha_{m+1}, \ldots, \alpha_n)$$

Since α establishes $(x = v)$ for α', none of the operators $\alpha_{k+1}, \ldots, \alpha_{m-1}$ changes x. Observe that in the plan Π

- the operators $\alpha_1, \alpha_2, \ldots, \alpha_{k-1}$ are possibly before α and α':

$$(\forall i \in [1..(k-1)])\ \neg(\alpha \prec \alpha_i) \text{ and } \neg(\alpha' \prec \alpha_i)$$

- the operators $\alpha_{m+1}, \ldots, \alpha_n$ are possibly after α and α':

$$(\forall i \in [(m+1)..n])\ \neg(\alpha_i \prec \alpha) \text{ and } \neg(\alpha_i \prec \alpha')$$

- since α' is not backward justified, by the choice of α we have:

$$\neg(\alpha \prec \alpha')$$

Therefore there exists another linearization $\overline{\Pi}'$ of Π, such that in this linearization

$$\alpha_1 \prec \alpha_2 \prec \ldots \prec \alpha_{k-1} \prec \alpha' \prec \alpha \prec \alpha_{m+1} \prec \ldots \prec \alpha_n$$

We are concerned with the value of x in this linearization, and we do not care about positions of the operators $\alpha_{k+1}, \ldots, \alpha_{m-1}$, which do not change x. Since $(x = v)$ is a precondition of α', and $\overline{\Pi}'$ is correct as a linearization of the correct plan Π, $(x = v)$ holds before α' in $\overline{\Pi}$. This means that after executing the sequence of the operators $\alpha_1, \alpha_2, \ldots, \alpha_{k-1}$ starting from the initial state, the value of x becomes v. Since the same operators are executed before α in $\overline{\Pi}$, the value of x before α is also v.

Thus, we have shown that if α establishes $(x = v)$ for some operator, then $(x = v)$ holds before α. By the definition of well-justification this means that α is not well-justified. □

Observe that while every well-justified plan is backward justified, this is not so for operators in a plan. A well-justified operator may not be backward justified. For example, suppose you have an empty cup and your goal is to fill it with water. The following plan is correct and achieves the goal (even though it is not optimal):

1. Pour water into the cup.
2. Pour water into a glass.
3. Empty the glass.

The second and third operators in this plan are not backward justified, since they do not contribute anything into achieving the goal. However, *operator 2* is well-justified, because upon its removal *operator 3* becomes illegal.

Note that there might be several distinct well-justified subplans of the same plan. For example, suppose one has a kettle of cold water, and needs a cup of hot water. The following plan leads to the desired result.

1. Boil water by putting the kettle onto a stove.
2. Pour the water into the cup.
3. Put the cup into a microwave.

This plan is not well-justified, because either the first or third operator may be skipped without violating the correctness of the plan. Thus, the plan has two well-justified subplans: one of them consists of the first two operators, and the other consists of the last two.

A simple algorithm that finds a well-justified subplan of a given plan is shown in Table 3.2. Here the algorithm *Legal_Plan* checks whether a plan is legal, that is whether the preconditions of all operators are satisfied before their execution. For each operator α, the algorithm calls the procedure *Legal_Operator* to check whether the operator α is legal. The algorithm *Legal_Operator* considers each precondition $(x = v)$ of α, and checks whether this precondition holds before α. Lines 2b–8b of the algorithm check whether some operator before α establishes $(x = v)$, or whether $(x = v)$ holds in the initial state. If $(x = v)$ does *not* hold in the initial state, and no operator before α establishes it, we conclude in line 9b that the plan is not correct. If $(x = v)$ is established, we need to make sure that no operator removes this value. This is done in lines 10b–17b: for each operator α_1 that may occur

before α and establishes a value of x different from v, we check if there exists an operator between α_1 and α that reestablishes $(x = v)$. If such an operator is not found, we conclude in line 17b that the plan is not correct. It is straightforward to verify that the running time of the algorithm *Legal_Operator* is $O(|Pre(\alpha)| \cdot |\Pi|^2)$, and the running time of the algorithm *Legal_Plan* is $O(P \cdot |\Pi|^2)$, where $P = \sum_{\alpha \in O} |Pre(\alpha)| + |S_g|$.

The algorithm *Goal_Achieved* checks whether the goal is satisfied in the final state. It is similar to the algorithm *Legal_Operator*: we just consider the goal as the preconditions of some "virtual" operator added at the end of the plan and check whether this preconditions are satisfied. The running time of this algorithm is $O(|S_g| \cdot |\Pi|^2)$.

To find a well-justified subplan of a given plan, the algorithm *Well_Justification* tries to remove each operator from the plan and then checks whether the remaining plan is still correct. (The notation "$\Pi - \{\alpha\}$" in the line 3 refers to the plan Π with the operator α removed.) If such an operator is found, the algorithm removes it and starts from the beginning with this new, shorter plan. Since it may remove an operator at most $|\Pi|$ times, and it calls the procedures *Legal_Plan* and *Goal_Achieved* at most $|\Pi|$ times between removals of operators, the total running time of the algorithm is $O(P \cdot |\Pi|^4)$.

3.3 Perfect justification

While well-justified plans cannot contain unnecessary operators, they still may contain unnecessary *groups of operators*. This means that while no single operator may be eliminated from the plan, several operators may be eliminated *together*. For example, consider the following plan of boiling water:

1. Fill a cup with water.
2. Empty the cup.
3. Fill the cup with water again.
4. Put the cup into a microwave.

This plan is well-justified: we cannot skip *operator 2*, because then we could not fill the cup again; and we cannot skip *operator 3*, because the cup has to be full when we put it into a microwave. However, we may skip *operators 2* and *3* together. To formalize this observation, we introduce the notion of a *perfectly justified* plan.

Intuitively, a plan is perfectly justified if no subset of its operators may be removed from the plan. In other words, this is the "best possible" justification.

Definition 3.4 (Perfect justification) *A correct plan* (S_0, S_g, Π) *is called* perfectly justified *if it does not have any correct proper subplan.*

Just by definition perfect justification is stronger than all justifications discussed above. Unfortunately, a perfect justification of a given plan cannot be found in polynomial time. The next theorem shows that the task to find a perfect justification of a given plan is NP-hard, even for linear plans. Moreover, it is NP-hard to check whether a linear plan is perfectly justified.

```
Well_Justification(S_0, S_g, Π)
1. repeat
2.     for each α ∈ Π do
3.         if Legal_Plan(S_0, Π − {α}, S_g) and Achieves_Goal(S_0, Π − {α}, S_g)
4.             then remove α from Π
5. until no operator is removed during the last execution of the loop

Legal_Plan(S_0, Π)
1a. for each α ∈ Π do
2a.     if not Legal_Operator(S_0, Π, α)
3a.         then /* Π contains illegal operator */ return(False);
4a. return(True)

Legal_Operator(S_0, Π, α)
1b.  for each (x = v) ∈ Pre(α) do
     begin
2b.      Established := False;
3b.      if (x = v) ∈ S_0
4b.          then Established := True;
5b.      for each α_1 ≺ α do
6b.          if (x = v) ∈ α_1
7b.              then Established := True;
8b.      if Established=False
9b.          then return(False);
10b.     for each α_1 such that ¬(α ≺ α_1)
11b.         if α_1 establishes a value of x different from v
                 then begin
12b.                     Reestablished := False;
13b.                     for every α_2 such that α_1 ≺ α_2 ≺ α do
14b.                         if (x = v) ∈ Eff(α_2)
15b.                             then Reestablished := True;
16b.                     if Reestablished = False
17b.                         then return(False)
                 end
     end;
18b. return(True)
```

Table 3.2: Finding a well-justified subplan of a given plan

Theorem 3.4 *Suppose we are given a linear plan (S_0, S_g, Π), and we wish to determine whether this plan is perfectly justified. This problem is NP-complete.*

Proof. The problem is trivially NP, since, given a subplan of a given plan, we may check in polynomial time whether this subplan is correct and achieves the goal (see the algorithms *Legal_Plan* and *Goal_Achieved* in the previous section). So, we need only to show that the problem is NP-hard.

We prove this claim by reducing 3-CNF-SAT problem to our problem. CNF is a conjunctive normal form with at most 3 logical variables in every clause. The 3-CNF-SAT problem is the problem to determine whether a given 3-CNF has a satisfying assignment. This problem is known to be NP-complete (see, for example, [Cormen *et al.*, 1990]).

In the proof below we use the literal representation of the problem domain. In other words, we reduce 3-CNF-SAT to the plan in the problem domain where all variables of the set $\mathcal{X}$ may accept only two values: *True* and *False*. In the proof we deal with two kinds of variables, both logical: the variables in the 3-CNF and the variables in the planning domain to which we reduce 3-CNF-SAT. To avoid confusion, we call the former *CNF-variables* and denote them by upper-case letters. On the other hand, variables in the planning domain are called *planning variables* and denoted by lower-case letters.

Suppose we are given 3-CNF with n distinct logical variables $X_1, X_2, \ldots, X_n$, and k distinct clauses $C_1, C_2, \ldots, C_k$. For each CNF-variable X_i we introduce two planning variables, v_i^+ and v_i^-. For each clause C_j we introduce a corresponding planning variable c_j. Finally, for each pair (X_i, C_j), where X_i is a CNF-variable *in* the clause C_j, we introduce a planning variable x_{ij}. We consider a problem domain that contains all introduced planning variables. That is, the set of planning variables in our problem domain is

$$\mathcal{X} = \bigcup_{i=1}^{n} \{v_i^+, v_i^-\} \cup \bigcup_{i=1}^{k} \{c_j\} \cup \{x_{ij} \mid X_i \text{ is in } C_j\}$$

Now we need several operators that change values of our planning variables. For each $i \in [1..n]$ we introduce an operator α_i with effects $(v_i^+ = \mathit{True}), (v_i^- = \mathit{False})$ and without any preconditions (see Table 3.3). For each pair (X_i, C_j), where X_i is a CNF-variable in the clause C_j, *without* negation ($\neg$) in front of this CNF-variable, we introduce the operator β_{ij} with a precondition $(v_i^+ = \mathit{True})$, and with effects $(c_j = \mathit{True}), (x_{ij} = \mathit{True})$. For each pair (X_i, C_j), where X_i is a CNF-variable in the clause C_j, *with* negation ($\neg$) in front of this CNF-variable, we introduce the operator γ_{ij} with a precondition $(v_i^- = \mathit{True})$, and with effects $(c_j = \mathit{True}), (x_{ij} = \mathit{True})$. Finally, we introduce the operator δ with preconditions $(v_1^- = \mathit{False}), (v_2^- = \mathit{False}), \ldots, (v_n^- = \mathit{False})$, whose effects are all $(v_i^- = \mathit{True})$'s and all $(x_{ij} = \mathit{False})$'s. All introduced operators are presented in Table 3.3.

We define an initial state as follows

- $(\forall i \in [1..n])\ v_i^- = \mathit{True}$ and $v_i^+ = \mathit{False}$
- $(\forall j \in [1..k])\ c_j = \mathit{False}$
- for all planning variables x_{ij} in our domain, $x_{ij} = \mathit{True}$

The goal of our plan is:

operators	preconditions	effects
α_i $(\forall i \in [1..n])$	—	$(v_i^+ = True), (v_i^- = False)$
β_{ij} (for each $X_i \in C_j$)	$(v_i^+ = True)$	$(c_j = True), (x_{ij} = True)$
γ_{ij} (for each $\neg X_i \in C_j$)	$(v_i^- = True)$	$(c_j = True), (x_{ij} = True)$
δ	$(v_1^- = False), \ldots, (v_n^- = False)$	$(v_1^- = True), \ldots, (v_n^- = True)$ and all $(x_{ij} = False)$'s

Table 3.3: Operators in the proof of NP-completeness

- $(\forall j \in [1..k])\ c_j = True$
- for all planning variables x_{ij} in our domain, $x_{ij} = True$

Now we present a linear plan that solves the goal:

$$\overline{\Pi} = (\alpha_1, \alpha_2, \ldots, \alpha_n, \delta, \text{ all } \beta_{ij}\text{'s in any order, all } \gamma_{ij}\text{'s in any order})$$

First we show that this is indeed a correct plan that solves the goal. All α-operators are certainly legal because they do not have any preconditions. The operator δ has preconditions $(v_1^- = False), (v_2^- = False), \ldots, (v_n^- = False)$, and each precondition $(v_i = False)$ is achieved by the corresponding α_i. For each operator β_{ij}, its precondition $(v_i^+ = True)$ is established by the corresponding operator α_i, and the value of v_i cannot be changed between α_i and β_{ij} neither by other α's, nor by other β's, nor by δ. For each operator γ_{ij}, its precondition $(v_i^- = True)$ is established by the operator δ and cannot be deleted by any of the following operators. Finally, the goal values, which are all $(c_j = True)$'s and all $(x_{ij} = True)$'s, are established by β_{ij} and γ_{ij}-operators.

Now we claim that the initial conjunctive normal form has a satisfying assignment if and only if the described plan has a correct proper subplan that solves the goal.

First assume that the conjunctive normal form has a satisfying assignment. W.l.o.g. we may assume that the satisfying assignment is

$$X_1 = X_2 = \ldots = X_m = True$$
$$X_{m+1} = X_{m+2} = \ldots = X_n = False$$

for some $m \in [0..n]$. Then we claim that the subplan

$$\overline{\Pi}' = (\alpha_1, \alpha_2, \ldots, \alpha_m, \text{ all } \beta_{ij}\text{'s such that } i \in [1..m], \text{ all } \gamma_{ij}\text{'s such that } i \in [(m+1)..n])$$

of $\overline{\Pi}$ is correct. The operators $\alpha_1, \alpha_2, \ldots, \alpha_m$ in $\overline{\Pi}'$ are legal just because they do not have preconditions. For each operator β_{ij}, $i \in [1..m]$, its precondition $(v_i^+ = True)$ is established by the operator α_i. For each γ_{ij}, $i \in [(m+1)..n]$, its precondition $(v_i^- = True)$ holds in the initial state, and since the operator α_i is not in the plan $\overline{\Pi}'$, no operator changes v_i^-. All x_{ij}'s have the value $True$ after execution of $\overline{\Pi}'$ because they have this value in the initial state, and no operator of $\overline{\Pi}'$ changes any of them. Finally, we need to show that all c_j's have the value $True$ in the final state. Consider some planning variable c_j, $j \in [1..m]$. Since we

consider a satisfying assignment of the conjunctive normal form, the clause C_j must have the value *True* under this assignment, and therefore it either contains some CNF-variable X_i *without* negation ($\neg$) in front of it such that $X_i = True$, or it contains some CNF-variable X_i *with* negation ($\neg$) such that $X_i = False$.

Case 1: X_i *without negation,* X_i = True. Then there is an operator β_{ij} in the plan $\overline{\Pi}'$. This operator establishes ($c_j = True$), and neither other β-operators, nor γ-operators changes c_j after it becomes *True*.

Case 2: X_i *with negation,* X_i = False. Then there is an operator γ_{ij} in the plan $\overline{\Pi}'$. This operator establishes ($c_j = True$), and other γ-operators do not change the value of c_j.

Now assume that the plan (S_0, S_g, Π) has a correct proper subplan. We need to show that the conjunctive normal form has a satisfying assignment. If we remove some operator α_i from $\overline{\Pi}$, then the precondition ($v_i^- = False$) of δ is no longer satisfied, and therefore we have to remove δ. On the other hand, if we have removed some operator β_{ij} or γ_{ij}, and have not removed δ, then δ establishes ($x_{ij} = False$), and no operator after δ establishes ($x_{ij} = True$), and therefore the resulting plan does not achieve the goal. Thus, whatever operator we remove from $\overline{\Pi}$, we have to remove δ too. This means that no correct proper subplan of our plan may contain δ. Thus, any correct proper subplan of $\overline{\Pi}$ has the form

$$\overline{\Pi}_1 = (\alpha_{k_1}, \alpha_{k_2}, \ldots, \alpha_{k_m}, \text{ some sequence of } \beta_{ij}\text{'s, some sequence of } \gamma_{ij}\text{'s})$$

We claim that if this is a correct plan, then the assignment

$X_{k_1} = X_{k_2} = \ldots = X_{k_m} = True$,
and all other CNF-variables $X_{k_{m+1}} = X_{k_{m+2}} = \ldots = X_{k_n} = False$

is a satisfying assignment of the conjunctive normal form. To prove this, we need to show that each clause C_j has the value *True*. Pick any $j \in [1..k]$. Since $\overline{\Pi}_1$ achieves the goal, one of its operators must establish ($c_j = True$). This may be done either by an operator β_{ij} (where i is arbitrary) or by an operator γ_{ij}.

Case 1: (c_j = True) *is achieved by* β_{ij}. By construction of the problem domain, existence of the operator β_{ij} implies that the CNF-variable X_i, *without* negation in front of it, is in the clause C_j. Some operator in $\overline{\Pi}_1$ must establish the precondition ($v_i^+ = True$) of β_{ij}. The only operator that achieves this precondition is α_i. Therefore, $\alpha_i \in \overline{\Pi}_1$, and therefore in our truth assignment $X_i = True$. Since C_j has the CNF-variable X_i without negation, and the value of $\mathcal{X}$ is *True*, the value of the clause C_j is also *True*.

Case 2: (c_j = True) *is achieved by* γ_{ij}. By construction of the problem domain, existence of the operator γ_{ij} implies that the CNF-variable X_i, *with* negation in front of it, is in the clause C_j. The precondition of γ_{ij} is ($v_i^- = True$). If the operator α_i is in the plan $\overline{\Pi}_1$, then it establishes the value ($v_i^- = False$), and no operator after it establishes ($v_i^- = True$), and so the precondition of γ_{ij} is not satisfied. Thus, α_i *cannot* be in the plan $\overline{\Pi}_1$, and therefore in our truth assignment $X_i = False$. Since $(\neg X_i) \in C_j$, and $X_i = False$, the value of the clause C_j is *True*. □

Corollary 3.1 *The problem to find a perfectly justified subplan of a given plan is NP-complete.*

3.4 Greedy justification

While the task of finding the best possible justified plan is NP-hard, one can design a greedy algorithm that finds an "almost" perfect justification. To check "usefulness" of some operator α in a plan Π, the algorithm proceeds as follows. First, it removes an operator α from the plan. After α has been removed, some operators of Π may become illegal, which means that now their preconditions are *not* satisfied before their execution. The algorithm removes the first illegal operator of the plan Π. If Π is a nonlinear plan, the algorithm removes every operator which is the first illegal operator in some linearization of Π. Then the algorithm examines the resulting plan, finds the illegal operators, and again removes all earliest illegal operators. The algorithm repeats this step until either all the remaining operators are legal, or no operators are left at all. If the remaining plan still solves the goal, then the initially removed operator α was not useful, and we say that α is not *greedily justified.*

The description of the algorithm is presented in Table 3.4. Here the algorithm *Illegal_Operators* finds the set of all illegal operators of the plan. It applies the procedure *Legal_Operator* (see Table 3.2) to every operator of the plan to check whether the operator is legal. Recall that the running time of the algorithm *Legal_Operator* is $O(|Pre(\alpha)| \cdot |\Pi|^2)$, and therefore the running time of the algorithm *Illegal_Operators* is $O(P \cdot |\Pi|^2)$, where $P = \sum_{\alpha \in O} |Pre(\alpha)| + |S_g|$.

It is easy to see that an operator is the first illegal operator in one of the linearizations of Π if and only if it is a minimal element in the set of illegal operators under the time-precedence relation $\prec_\Pi$. So, we use the algorithm *Minimal_Elements* (described in Section 2.6) in line 4 to find the earliest illegal operators. After the earliest illegal operators are found, we remove them from the plan (line 5). We repeat this operation until the plan does not contain illegal operators.

Then the algorithm *Goal_Achieved* (described in the previous section) checks whether a plan achieves the goal. It takes $O(|S_g| \cdot |\Pi|^2)$ time.

Observe, that we perform the removal of illegal operators at most $|\Pi|$ times. Since the running time of finding the set of the earliest illegal operators between removals is $O(P \cdot |\Pi|^2)$, the overall running time of the algorithm is $O(P \cdot |\Pi|^3)$.

As an example, we consider our water-boiling plan:

1. Fill the cup with water.
2. Empty the cup.
3. Fill the cup with water again.
4. Put the cup into a microwave.

We wish to check whether *operator 2* is greedily justified. We start by removing this operator from the plan. Now *operator 3* is illegal, because we cannot fill a cup which is already full. So, we remove *operator 3* too. We are left with the plan

1. Fill the cup with water.
4. Put the cup into a microwave.

which is correct and solves the goal. Thus, *operator 2* in the initial plan is not greedily justified.

Greedy_Justify_Checking(S_0, S_g, Π, α)
1. remove α from Π;
2. **repeat**
3. *Illegals* := Illegal_Operators(S_0, S_g, Π);
4. *Earliest_Illegals* := Minimal_Elements(*Illegals*);
5. remove all operators of the set *Earliest_Illegals* from Π
6. **until** /* Π does not contain illegal operators */ *Illegals*=$\emptyset$;
7. **if** /* plan still achieves the goal */ Goal_Achieved(S_0, S_g, Π)
8. **then** return(Π) /* Π is a correct subplan of the initial plan */
9. **else** return("not found") /* α in the initial plan is greedily justified */

Illegal_Operators(S_0, S_g, Π)
1a. *Illegals* := $\emptyset$;
2a. **for** each $\alpha \in \Pi$ **do**
3a. **if not** Legal_Operator(S_0, Π, α)
4a. **then** *Illegals* := *Illegals* $\cup \{\alpha\}$;
5a. return(*Illegals*)

Table 3.4: Checking if the operator α in the plan Π is greedily justified

Lemma 3.2 *If an operator is greedily justified, it is also well-justified.*

Proof. Assume that an operator α is not well-justified. If we use the algorithm *Greedy_Justify_Checking* to check the usefulness of α, then α is removed at the first step of execution, and, by Lemma 3.1, the remaining plan is correct and solves the goal. So α is not greedily justified. □

A plan is said to be greedily justified if all its operators are greedily justified. It follows from the above lemma that such a plan is always well-justified. An algorithm that finds a greedily justified subplan of the plan (S_0, S_g, Π) is presented in Table 3.5. To find a greedily justified subplan of a given plan, the algorithm uses the procedure *Greedy_Justify_Checking* to check whether each operator of the plan is greedily justified. If a non-greedily-justified operator is found, the algorithm replaces the initial plan with its subplan found by *Greedy_Justify_Checking*, and starts from the beginning with this new, shorter plan. Since it may remove operators from the initial plan Π at most $|\Pi|$ times, and it calls the procedure *Greedy_Justify_Checking* at most $|\Pi|$ times between removals of operators, the total running time of the algorithm is $O(P \cdot |\Pi|^5)$.

The running time may be considerably improved in the case of a linear plan. The algorithm that finds a greedily justified subplan of a linear plan is shown in Table 3.6. To determine whether some operator α is greedily justified, the algorithm removes this operator and executes the remaining operators in order. If an illegal operator is encountered, the algorithm removes this operator and continues to execute the plan. Thus, it removes all illegal operators and receives the final state that the plan achieves with the illegal operators removed. If the goal is not achieved, then the initially removed operator α is greedily justified. On the other hand, if the new plan achieves the goal, then it is an optimized

```
Greedy_Justification
1. repeat
2.     Something_Removed := False;
3.     for each α ∈ Π do
       begin
4.       Π' := Greedy_Justify_Checking(S_0, S_g, Π, α);
5.       if /* α is not greedily justified */ Π' ≠ "not found"
            then begin
7.                   Π := Π';
8.                   Something_Removed := True
                 end
       end
9. until Something_Removed = False
```

Table 3.5: Finding a greedily justified subplan of a nonlinear plan

version of the initial plan. Then we apply our algorithm recursively to check if this new, shorter plan is greedily justified.

Recall that the running time of the procedure *Check_Preconditions* is $O(|Pre(\alpha)|)$ or $O(|Pre(\alpha)| \cdot \log |S|)$, depending on representation of S (see Section 2.6), and the running time of *Apply* is either $O(|Eff(\alpha)|)$ or $O(|Eff(\alpha)| \cdot \log |S|)$. Within the loop of lines 4–7, the algorithm executes *Check_Preconditions* and *Apply* at most once for each operator. So the running time of this loop for the full representation of S is

$$\sum_{\alpha_1 \in \Pi} (|Pre(\alpha)| + |Eff(\alpha)|) = O(P + E)$$

The loop in lines 4–7 is executed at most once for each operator of the plan *before* the recursive call, and so the running time of the algorithm before the recursive call is $O((P + E) \cdot |\overline{\Pi}|)$. Finally, during each execution of the recursive call, $\overline{\Pi}$ is reduced by at least one operator, and therefore the depth of the recursion is at most $|\overline{\Pi}|$. Thus, the total running time is

$$O((P + E) \cdot |\overline{\Pi}|^2)$$

Similarly we may show that the running time in the case of the closed-world representation of S is $O((P + E) \cdot |\overline{\Pi}|^2 \cdot \log |S_n|)$, where S_n is the final state of $\overline{\Pi}$.

3.5 A spectrum of justified plans

Table 3.7 presents different kinds of justification and running time necessary to find a justified subplan of a plan for each kind of justification. Running time is presented for algorithms dealing with nonlinear plans. Recall that the algorithm that finds a greedily justified version of a linear plan is much faster; it takes only $O((P + E) \cdot |\overline{\Pi}|^2 \cdot |S_n|)$ time.

The table may be viewed as a spectrum of justified plans. On one end of the spectrum plans are backward justified. A backward justified subplan of a given plan is not hard to

Linear_Greedy_Justification($S_0, S_g, \overline{\Pi}$)
1. **for** each $\alpha \in \overline{\Pi}$ **do**
   begin
2. $\overline{\Pi}_1 := \overline{\Pi} - \{\alpha\}$;
3. $S := S_0$;
4. **for** α_1:= (first operator of $\overline{\Pi}_1$) **to** (last operator of $\overline{\Pi}_1$) **do**
5. **if** /* α_1 is legal */ Check_Preconditions(S, α_1)
6. **then** /* execute α_1 */ $S :=$ Apply(S, α_1)
7. **else** /* remove α_1 from $\overline{\Pi}_1$ */ $\overline{\Pi}_1 := \overline{\Pi}_1 - \{\alpha_1\}$;
8. **if** $S_g \subseteq S$ /* $\overline{\Pi}_1$ achieves the goal S_g */
9. **then** return(Linear_Well_Justification($\overline{\Pi}_1, S_0, S_g$))
   end;
10. return($\overline{\Pi}$)

Table 3.6: Finding a greedily justified subplan of a linear plan

kind of subplan	**running time to find it**	
perfectly justified	NP-complete	stronger justification
greedily justified	$O(P \cdot \lvert\Pi\rvert^5)$	↑
well-justified	$O(P \cdot \lvert\Pi\rvert^4)$	↓
backward justified	$O(E \cdot \lvert\Pi\rvert^2)$	weaker justification

Table 3.7: Kinds of justified subplans and running time to find them

find, but it may contain some "useless" operators. The other end of the spectrum contains perfectly justified plans. They cannot have any useless operators, but it is NP-hard to find a perfectly justified subplan of a given plan.

Chapter 4

Ordered and Semi-ordered Abstraction Hierarchies

Hierarchical problem solving uses abstraction to reduce the complexity of the search by dividing up a problem into smaller subproblems. Given a problem space and a hierarchy of abstractions, a hierarchical problem solver first solves a problem in an abstract space, and then refines it in successively more detailed spaces. The use of abstraction in problem solving is an effective approach to reducing search, but finding good abstraction is a difficult problem.

ABSTRIPS [Sacerdoti, 1974] was the first attempt to create abstraction hierarchies automatically. [Knoblock, 1990] describes an abstraction learner, called ALPINE, that completely automates the formation of abstraction spaces. The theoretical ideas behind Knoblock's planner are based on the notion of *ordered hierarchies* [Knoblock *et al.*, 1991], which characterizes some intuitions behind "good" abstraction hierarchies. This notion guarantees that *every* refinement of an abstract plan leaves the abstract plan structurally unchanged. The notion of ordered hierarchies is defined via justified plans, described in the previous chapter. The results presented in [Knoblock *et al.*, 1991] are based on backward justified plans, but they may be defined via any other kind of justification as well. The conditions presented in [Knoblock *et al.*, 1991] for generating ordered hierarchies are sufficient for a hierarchy to be ordered, but not necessary.

The main problem with Knoblock's algorithm is that a hierarchy generated by the algorithm often has too few levels of abstraction, or even collapses into a single level. The purpose of this chapter is to present several methods for increasing the number of levels of an abstraction hierarchy, while still preserving the ordered property. We approach the problem of increasing the number of abstract levels in several small steps. Each single step does not provide a considerable improvement by itself, but small improvements introduced at every step are accumulated, and together they enable us to generate a hierarchy with more levels than those generated by Knoblock's algorithm in many problem domains.

At the first step we slightly relax the restriction imposed on hierarchies by Knoblock's definition of an ordered abstraction hierarchy. Knoblock's definition states that no operator α, inserted during refinement process at the concrete level of abstraction, may have an abstract-level effect. We, on the other hand, allow such a low-level operator α to establish a

value of an abstract-level variable. However, we demand that all operators with abstract-level effects inserted during refinement process may be removed after completing the refinement process without violating the correctness of the refined plan. A hierarchy that satisfies our requirement is called *semi-ordered.*

With this relaxation an abstract hierarchy may have more abstract levels then Knoblock's ordered hierarchy. We show that, using a semi-ordered hierarchy instead of an ordered one, a planner is still able to generate *ordered* refinements of abstract-level plans.

At the second step we present *necessary and sufficient* conditions for an abstraction hierarchy to be semi-ordered, which are less restrictive than Knoblock's sufficient conditions, and allow us to increase the number of abstraction levels, since they impose less constraints onto values of criticalities.

Finally, we show that the hierarchies defined via well-justified or greedily justified plans can be finer-grained than the hierarchies defined via backward justification. We present a sufficient condition for such hierarchies and a learning algorithm that generates hierarchies based on this condition.

4.1 Previous work

In this section we describe the definition of the ordered abstraction hierarchy and sufficient conditions for a hierarchy to be ordered, presented in [Knoblock *et al.*, 1991]. Knoblock presented his definitions and theorems for the literal-representation of problem domains. In this chapter we restate his results for the variable representation. This restatement does not violate the ideas behind his results, nor the correctness of his proofs.

Let Π' be a justified abstract-level plan. Intuitively, Π is the refinement of Π' at a concrete level of an abstraction hierarchy if all operators and their ordering relations of Π' are preserved in Π, and the new operators have been inserted for the purpose of satisfying concrete-level preconditions.

Definition 4.1 (Refinement)
Let (S_0, S_g, Π') *be a correct plan at level* $(i+1)$. (S_0, S_g, Π) *is a refinement of* (S_0, S_g, Π') *at level* i *if*

1. (S_0, S_g, Π) *is a correct plan at level* i, *and*
2. *there is a one-to-one function* c *mapping each operator of* Π' *into* Π, *such that*
   (a) $\forall \alpha \in \Pi'$, $c(\alpha) = \alpha$, *and*
   (b) if $\alpha_1 \prec_{\Pi'} \alpha_2$, *then* $c(\alpha_1) \prec_{\Pi} c(\alpha_2)$

We now consider a property that ensures that operators added at some level of abstraction in the refinement process do not change higher-level variables. A refinement that satisfies this property is called an *ordered refinement.*

Definition 4.2 (Backward ordered refinement)
Let (S_0, S_g, Π') *be a correct backward justified plan at level* $(i+1)$. (S_0, S_g, Π) *is a* backward ordered refinement *of* (S_0, S_g, Π') *at level* i *if*

1. (S_0, S_g, Π) *is a refinement of* (S_0, S_g, Π'),
2. (S_0, S_g, Π) *is backward justified at level* i, *and*
3. $\forall \alpha \in \Pi$, *if* $\alpha \notin \Pi'$, *then for any variable* x *in the effects of* α, $crit(x) \leq i$.

The third condition states that the newly inserted operators do not achieve a value of any variable whose criticality is higher than i.

Definition 4.3 (Backward ordered hierarchy)
An abstraction hierarchy is backward ordered *if any backward justified refinement of any correct backward justified plan is a backward ordered refinement.*

The backward ordered hierarchy ensures that the values of abstract-level variables are never changed by any operator while planning at lower levels of abstraction. Intuitively, it characterizes "good" hierarchies, which allow us to increase the efficiency of planning [Knoblock, 1991a]. (In Knoblock's paper such hierarchies are called *"ordered"*, without the word "backward".)

The following restrictions imposed onto an abstraction hierarchy are sufficient but not necessary to guarantee the backward ordered property of the hierarchy.

Restrictions 1 and 2 *Let* O *be the set of operators in a domain.* $\forall \alpha \in O$, $\forall x \sqsubset \mathrm{Pre}(\alpha)$, *and* $\forall x_1, x_2 \sqsubset \mathrm{Eff}(\alpha)$

1. $crit(x_1) = crit(x_2)$, *and*
2. $crit(x) \leq crit(x_1)$.

Stated simply, all effects of an operator have the same criticality, and their criticality is at least as great as the criticalities of the preconditions of the operator.

Theorem 4.1 *Any abstraction hierarchy satisfying Restrictions 1 and 2 is a backward ordered hierarchy.*

The proof of the theorem, the algorithm based on this theorem that generates an abstraction hierarchy, and the description of a planner that uses a backward ordered abstraction hierarchy may be found in [Knoblock, 1991a]. While planning in a backward ordered abstraction hierarchy, the planner refines only backward ordered plans. Thus, after finding a correct plan at some level of abstraction, the planner first finds a backward justified version of this plan, and then refines it at a lower level of abstraction.

4.2 Backward semi-ordered hierarchies

4.2.1 Definition and properties of semi-ordered hierarchies

In the previous section we defined an ordered refinement Π of an abstract plan Π' as such a refinement that none of the newly inserted operators has an abstract-level effect. We slightly relax this requirement to define a *semi-ordered refinement*. According to this relaxed definition, an operator α, inserted into Π while refining Π at a lower level, *may* have abstract-level effects, but all newly inserted operators with abstract-level effects can be removed together from Π without violating the correctness of Π.

Operators inserted during a refinement process that have abstract-level effects are called *abstract-effect operators.*

Definition 4.4 (Abstract-effect operators)
Let Π' *be a backward justified plan at level* $(i+1)$ *of abstraction, and* Π *be a refinement of* Π *at the* i*-th level. Let* α *be an operator of* Π *which is not an operator of* Π'*.* α *is said to be an* abstract-effect *operator if there exists* $x \sqsubset \mathrm{Eff}(\alpha)$ *such that* $\mathrm{crit}(x) > i$*.*

A refinement Π of an abstract-level plan Π' is called *semi-ordered* if all abstract-level operators may be removed from Π without violating the correctness of Π. Thus, while refining an (i+1)-level plan Π' at level i, we first find its correct backward justified refinement Π, and then we may remove all newly inserted operators of Π whose effects have criticalities higher than i.

Definition 4.5 (Backward semi-ordered refinement)
Let (S_0, S_g, Π') *be a correct backward justified plan at level* $(i+1)$*.* (S_0, S_g, Π) *is a* backward semi-ordered refinement *of* Π' *at level* i *if*

1. (S_0, S_g, Π) *is a refinement of* (S_0, S_g, Π')*,*
2. (S_0, S_g, Π) *is backward justified at level* i*, and*
3. (S_0, S_g, Π) *remains correct at level* i *upon the removal of all abstract-effect operators*

Observe that a semi-ordered refinement is ordered if and only if it does not contain abstract-effect operators. Our next lemma shows that any semi-ordered refinement of an abstract-level plan may be easily converted into an ordered refinement.

Lemma 4.1 *Let* Π' *be a correct backward justified plan at level* $(i+1)$*, and* Π *be a backward semi-ordered refinement of* Π' *at the* i*-th level. Let* Π_1 *be obtained from* Π *by removing all abstract-effect operators, and* Π_2 *be obtained from* Π_1 *by removing all operators that are not backward justified in* Π_1 *at the* i*-th level. Than* Π_2 *is a backward ordered refinement of* Π'*.*

Proof. By the definition of abstract-effect operators, none of the abstract-effect operators of Π belongs to Π', and therefore Π_1 contains all operators of Π'. Also, Π_1 is correct by the definition of a semi-ordered refinement. Thus, Π_1 is a refinement of Π'.

Next we wish to show that every operator α of the plan Π' is backward justified in the plan Π_1. To prove it, we notice that if α is backward justified in Π', it establishes either a goal literal or a precondition of another backward justified operator with the criticality at least $(i+1)$ (recall that Π' is an $(i+1)$-level plan). All newly inserted operators of the plan Π_1 are *not* abstract-effect operators, and therefore the criticalities of their effects are at most i. Thus, none of the newly inserted operators reestablishes an $(i+1)$-level precondition established by α, and therefore α is still backward justified.

Thus, the plan Π_2, obtained from Π_1 by removing all operators that are not backward justified at level i, still contains all operators of Π'. By Theorem 3.1, Π_2 is correct and backward justified at level i. Therefore Π_2 is a backward ordered refinement of Π'. □

We may remove all abstract-effect operators in $O(E)$ time, where $E = \sum_{\alpha \in \Pi'} |\mathit{Eff}(\alpha)|$, by checking all effects of all operators inserted into Π'. An algorithm that removes the

```
Remove_Abstract_Effects(Π', Π, i)
/* Π is an i-th level refinement of Π' */
1. for each α ∈ Π do
2.    if α ∉ Π'
        then begin
3.             Abstract_Effect := False;
4.             for each x ⊏ Eff(α) do
5.                if crit(x) > i
6.                   then Abstract_Effect := True;
7.             if Abstract_Effect = True  /* α is an abstract-effect operator */
8.                then remove α from Π
           end;
9. return(Π)
```

Table 4.1: Removing abstract-effect operators from a refinement

abstract-effect operators is shown in Table 4.1. Then it takes $O(E \cdot |\Pi|^2)$ to find a backward version of the resulting plan. So the total time of converting a semi-ordered refinement into an ordered refinement is $O(E \cdot |\Pi|^2)$.

Definition 4.6 (Backward semi-ordered hierarchy)
An abstraction hierarchy is called backward semi-ordered *if any backward justified refinement of any correct backward justified plan is a backward semi-ordered refinement.*

The ordered property of an abstraction hierarchy requires a refinement to be ordered, while the semi-ordered property requires a refinement to be only semi-ordered. Since every ordered refinement *is* semi-ordered, we conclude that the semi-ordered property of an abstraction hierarchy is *less* restrictive than the ordered property, and thus allows us to generate a hierarchy with more levels.

Observe that while planning in a backward ordered hierarchy, a planner may work with *ordered* (not semi-ordered) refinements. After finding a correct plan at some level of abstraction, the planner finds a backward justified version of this plan, and thus obtains a semi-ordered refinement. Then the planner removes abstract-effect operators and finds a backward justified version of the resulting plan. By Lemma 4.1, the resulting plan is an ordered refinement of the abstract-level plan. Thus, the planning process in a semi-ordered hierarchy may be briefly described as follows:

```
Refine(Π', i) {Π is a correct backward-justified plan at level i + 1}
1. if i + 1 = 0 {Π is a concrete-level plan}
2.    then return(Π);
3. Find an i-level backward justified refinement Π of Π';
4. Remove all abstract-effect operators from Π';
      Backtracking point: consider all such refinements.
5. Refine(Π, i − 1)
```

The running time of this double-justification is $O(E \cdot |\Pi|^2)$, that is the same as the running time of single backward justification during planning in an ordered hierarchy. Thus, the asymptotic running time of the planning algorithm remains the same.

4.2.2 Necessary and sufficient conditions of the semi-ordered property

To generate semi-ordered hierarchies that are finer-grained than Knoblock's ordered hierarchies, we need to find a set of restrictions that are sufficient for a hierarchy to be semi-ordered, but less restrictive than Knoblock's Restrictions 1 and 2. To do this, we first need to introduce the following definition.

Definition 4.7 *We say that an expression* $(x = v)$ forbids *an operator* α*, if for any state* S *in which the value of* x *is equal to* v*, no legal plan with the initial state* S *may contain* α*. Similarly, an expression* $(x \neq v)$ *forbids an* α*, if for any state* S *in which the value of* x *is different from* v*, no legal plan with the initial state* S *may contain* α*.*

Intuitively, $(x = v)$ forbids an operator α if once the value of x equals v, we can never achieve the preconditions of α, and we can never apply α. For example, the literal "no water available" forbids the operator "boil water". If $(x = v)$ is a precondition of an operator α, and $x \neq v$ forbids α, we call $(x = v)$ a *forbidding precondition* of α. The algorithm for determining which preconditions of an operator are forbidding is presented in the next subsection, but for now we assume that forbidding preconditions are known.

Before we apply the notion of forbidding preconditions to generating semi-ordered hierarchies, observe that this notion may be useful even in non-hierarchical planning. Most planning algorithms build plans by backward chaining from the goal state (see, for example, [Minton *et al.*, 1991]). Intermediate plans during backward chaining are incorrect, and a planning algorithm tries to achieve the correctness by inserting new operators to establish the preconditions of all operators currently in the plan, or by imposing time-precedence constraints onto operators that lead to establishing the preconditions of existing operators. However, if a *forbidding* precondition of an operator does not hold, the planner should not try to establish this precondition by inserting a new operator, since we know that once a forbidding precondition of an operator does not hold, the operator can never be applied. (The planner still may establish a forbidding precondition by imposing time-precedence constraints.) So, if a forbidding precondition does not hold, the algorithm should backtrack instead of trying to establish this precondition. Thus, we may reduce the search space by using a planning algorithm that does not try to establish forbidding operators and thus avoiding dead ends. We call such an algorithm a *forbidding-restricted* planner.

Now we introduce Restriction 2′, which is weaker than Restriction 2, but is still enough to guarantee that a hierarchy is semi-ordered.

Restrictions 1 and 2′ *Let* O *be the set of operators in a domain.* $\forall \alpha \in O$, $\forall x$ *such that* $(x = v) \in \mathrm{Pre}(\alpha)$*, and* $\forall x_1, x_2 \sqsubset \mathrm{Eff}(\alpha)$

1. $crit(x_1) = crit(x_2)$*, and*

2′. *if the expression* $(x = v)$ *is not a forbidding precondition of* α*,*

then $crit(x) \leq crit(x_1)$.

To see that Restriction 2′ is weaker than 2, observe that Restriction 2 requires that, for every pair of a precondition variable x and an effect variable x_1, $crit(x) \leq crit(x_1)$. Restriction 2′ requires that only non-forbidding preconditions satisfy this restriction.

Below we present the main theorem of this section that shows that Restrictions 1 and 2′ are *sufficient* conditions of the semi-ordered property. If a problem domain does not have initial-state axioms, these restrictions are also *necessary* for the semi-ordered property.

Theorem 4.2

- *Any abstraction hierarchy satisfying Restrictions 1 and 2′ is a backward semi-ordered hierarchy.*
- *If the problem domain does not have initial-state axioms, then any backward semi-ordered hierarchy satisfies Restrictions 1 and 2′.*

Informally, the possibility of replacing Restriction 2 with less restrictive 2′ may be explained as follows. Let $(x = v) \in Pre(\alpha)$, $(x_1 = v_1) \in Eff(\alpha)$, and the expression $(x \neq v)$ forbids α. Let Π be a backward refinement of an abstract plan. Suppose $(x_1 = v_1)$ does not hold, and we wish to achieve it by applying α. If $(x = v)$ does not hold at the same point in Π, then we can never apply α by the definition of *forbidding*. Therefore, no operator that changes x needs to be applied, and the ordered property holds. On the other hand, if $(x = v)$ holds at this point in Π, we do not need to achieve it, and so again we do not need to change any variable with criticality $crit(x)$. Thus, while achieving $(x_1 = v_1)$, we don't change any variable at a higher level of abstraction. Now we present a formal proof of the theorem.

Proof. Assume that Restrictions 1 and 2′ hold. Consider an arbitrary plan Π' at the $(i+1)$-th level of abstraction, and its backward justified refinement Π at the i-th level. We wish to show that Π is a semi-ordered refinement of Π', that is Π remains correct at level i upon the removal of all its abstract-effect operators. Let Π_1 be the plan obtained from Π by removing all abstract-effect operators. To show that Π_1 is correct, it is enough to proof that all linearizations of Π_1 are correct. So, we consider an arbitrary linearization $\overline{\Pi}_1$ of Π_1, and prove the correctness of this linearization at the i-th level.

Let $\overline{\Pi}'$ be a plan obtained from $\overline{\Pi}_1$ by removing all operators that do not belong to Π'. Then $\overline{\Pi}'$ is a linearization of Π', and therefore $\overline{\Pi}'$ is correct at level $(i+1)$ as a linearization of a correct plan. Let $\overline{\Pi}$ be a linearization of Π such that $\overline{\Pi}'$ is a subplan of $\overline{\Pi}$. (Such a linearization of Π exists by Lemma 2.2.) Notice that $\overline{\Pi}$ is correct at level i as a linearization of the correct plan Π.

To prove the correctness of $\overline{\Pi}_1$, we divide its operators into two groups: the operators of $\overline{\Pi}'$, and the operators inserted into $\overline{\Pi}'$ during the refinement process. We need to prove that the operators of the both groups are legal, and that the goal is satisfied in the final state.

Claim 1. Every operator α of $\overline{\Pi}_1$, such that $\alpha \notin \overline{\Pi}'$, is legal.

Consider an arbitrary precondition $(x = v)$ of α. We need to show that $(x = v)$ holds before α in $\overline{\Pi}_1$.

Case 1: $(x = v)$ is not a forbidding precondition. Observe that since α is a newly inserted non-abstract-effect operator, the criticality of its effects is at most i. Therefore, by Restriction 2', $crit(x) \leq i$. Since $\overline{\Pi}$ is correct, $(x = v)$ holds before α in $\overline{\Pi}$. $\overline{\Pi}_1$ is obtained from $\overline{\Pi}$ by the removal of the abstract-effect operators. Observe that by Restriction 1, *all* effects of every abstract-effect operator have the criticality higher than i. This means that no abstract-effect operator changes x, and therefore the value of x before α is the same in $\overline{\Pi}$ and $\overline{\Pi}_1$. Therefore, $(x = v)$ holds before α in $\overline{\Pi}_1$.

Case 2: $(x = v)$ is a forbidding precondition. Let us look at $\overline{\Pi}$. This plan is correct, and therefore $(x = v)$ holds in *all* states preceding α. (If $(x = v)$ does not hold at some state, then, by the definition of a forbidding precondition, α can never be applied after this state.) This means that $(x = v)$ holds in the initial state S_0, and no operator preceding α achieves any other value of x. Since $\overline{\Pi}_1$ is a subplan of $\overline{\Pi}$, we may conclude that no operator of $\overline{\Pi}_1$ preceding α achieves any value of x different from v. Since $\overline{\Pi}_1$ is executed from the same initial state S_0, in which $(x = v)$ holds, we conclude that $(x = v)$ holds before α in $\overline{\Pi}_1$.

Claim 2. Every operator α of $\overline{\Pi}_1$, such that $\alpha \in \overline{\Pi}'$, is legal.

Again we need to show that every precondition $(x = v)$ of α holds before α in $\overline{\Pi}_1$. Since $\overline{\Pi}_1$ is a plan at the i-th level of abstraction, we are concerned only with preconditions whose criticalities are no less than i.

Case 1: $crit(x) = i$. Since $\overline{\Pi}$ is correct, $(x = v)$ holds before α in $\overline{\Pi}$. Recall that $\overline{\Pi}_1$ is obtained from $\overline{\Pi}$ by removing abstract-effect operators, and by Restriction 1 *all* effects of every abstract-effect operator have the criticality higher than i. Thus, none of the abstract-effect operators change x, and therefore $(x = v)$ still holds before α upon the removal of those operators.

Case 2: $crit(x) > i$. Observe that $\overline{\Pi}_1$ is a refinement of $\overline{\Pi}'$ obtained by inserting *only* non-abstract-effect operators. In other words if α_1 is an operator of $\overline{\Pi}_1$ which is not an operator of $\overline{\Pi}'$, then the criticality of all effects of α is at most i. This means that if some operator of $\overline{\Pi}_1$ changes x, this operator is also an operator of $\overline{\Pi}'$. Thus, the value of x before α is the same in $\overline{\Pi}'$ and $\overline{\Pi}_1$. Since $\overline{\Pi}'$ is correct, we conclude that $(x = v)$ holds before α both in $\overline{\Pi}'$ and $\overline{\Pi}_1$.

Claim 3. $\overline{\Pi}_1$ achieves the goal.

The proof of this claim is the same as the proof of Claim 2: we just consider the goal as the set of preconditions of some special operator added at the end of $\overline{\Pi}'$.

Now we need to show that if a planning domain does not have initial-state axioms, then Restrictions 1 and 2' are *necessary* conditions of the semi-ordered property of an abstraction hierarchy.

First suppose that Restriction 1 does not hold, that is there exists an operator α with effects $(x_1 = v_1)$ and $(x_2 = v_2)$ such that $crit(x_1) < crit(x_2)$. We need to show that there exists some abstract-level plan whose refinement is not semi-ordered. We construct the initial state S_0 of our plan as follows. First take the empty set and add all preconditions of α to this set. If $x_1 \not\sqsubseteq Pre(\alpha)$, we choose a value v_1' of x_1 such that $v_1' \neq v_1$, and add $(x_1 = v_1')$ to S_0. If $x_2 \not\sqsubseteq Pre(\alpha)$, we choose a value v_2' of x_2, different from v_2, and add $(x_2 = v_2')$ to S_0. Formally, S_0 is defined as follows.

$$\text{let } v_1' \in (D(x_1) - \{v_1\}) \text{ and } v_2' \in (D(x_2) - \{v_2\})$$
$$\text{then } S_0 = \begin{cases} Pre(\alpha) & \text{if } x_1 \text{ and } x_2 \sqsubset Pre(\alpha) \\ Pre(\alpha) \cup (x_1 = v_1') & \text{if } x_1 \not\sqsubset Pre(\alpha) \text{ and } x_2 \sqsubset Pre(\alpha) \\ Pre(\alpha) \cup (x_2 = v_2') & \text{if } x_1 \sqsubset Pre(\alpha) \text{ and } x_2 \not\sqsubset Pre(\alpha) \\ Pre(\alpha) \cup (x_1 = v_1') \cup (x_2 = v_2') & \text{if } x_1 \text{ and } x_2 \not\sqsubset Pre(\alpha) \end{cases}$$

Recall that the same value of a variable cannot be both in the preconditions and effects of an operator, and therefore $(x_1 = v_1), (x_2 = v_2) \notin Pre(\alpha)$. Thus, in all four cases both x_1 and x_2 are specified in S_0, and their values are different from respectively v_1 and v_2:

$$x_1, x_2 \sqsubset S_0, \text{ and}$$
$$(1)\ S_0(x_1) \neq v_1$$
$$(2)\ S_0(x_2) \neq v_2$$

Also, by the construction of S_0, the operator α is legal in S_0:

$$Pre(\alpha) \subseteq S_0$$

Since our problem domain does not have initial-state axioms, S_0 may be the initial state of a plan. We consider a plan with the initial state S_0 that solves the goal $S_g = \{(x_1 = v_1)\}$. Let $i = crit(x_1)$ and $j = crit(x_2)$ (where $i < j$). On the j-th level of abstraction the goal is the empty set, and the empty plan $\Pi_j = ()$ is a backward justified plan that solves the goal. It is easy to check that the plan $\Pi_i = (\alpha)$ is a backward justified refinement of Π_j at the i-th level. Since $x_2 \sqsubset \mathit{Eff}(\alpha)$ and $crit(x_2) > i$, α is an abstract-effect operator, and after the removal of this operator our i-th level plan does *not* remain correct. Therefore Π_i is not a semi-ordered refinement of Π_j.

Next suppose that Restriction 2′ does not hold, that is there exists an operator α and values $(x = v) \in Pre(\alpha)$ and $(x_1 = v_1) \in \mathit{Eff}(\alpha)$ such that $crit(x_1) < crit(x)$ and $(x = v)$ is not a forbidding precondition of α. Again, we need to show that there exists some abstract-level plan whose refinement is not ordered. If Restriction 1 does not hold, then the hierarchy is not backward semi-ordered. So we assume that Restriction 1 holds.

Let $i = crit(x_1)$ and $j = crit(x)$ (where $i < j$). Since $(x = v)$ is not a forbidding precondition of α, there exists a correct plan Π' with an initial state S_0 such that the value of x is different from v in the initial state, $S_0(x) \neq v$, and Π' contains α. Let $\overline{\Pi}'$ be some linearization of Π', and α be the k-th operator in $\overline{\Pi}'$:

$$\overline{\Pi}' = (\alpha_1, \alpha_2, \ldots, \alpha_{k-1}, \alpha, \alpha_{k+1}, \ldots, \alpha_n)$$

We define $\overline{\Pi}$ as the part of $\overline{\Pi}'$ from the beginning to the operator α:

$$\overline{\Pi} = (\alpha_1, \alpha_2, \ldots, \alpha_{k-1}, \alpha)$$

This plan solves the goal $S_g = \{(x_1 = v_1)\}$. Let $\overline{\Pi}_i$ be the plan obtained from $\overline{\Pi}$ by the removal of the operators that are not backward justified at the i-th level of abstraction. By Theorem 3.1, $\overline{\Pi}_i$ is a correct backward justified plan at level i. Since the operator α in $\overline{\Pi}$ establishes the value $(x_1 = v_1)$ for the goal, it is backward justified, and therefore $\alpha \in \overline{\Pi}_i$. At level j the goal is empty, and the empty plan $\overline{\Pi}_j = ()$ is a backward justified plan that

achieves the goal at the j-th level. Clearly $\overline{\Pi}_i$ is a backward justified refinement of $\overline{\Pi}_j$. Let $\overline{\Pi}'_i$ be a plan obtained from $\overline{\Pi}_i$ by removing all abstract-effect operators. Observe that $crit(x) > i$, and therefore any operator of $\overline{\Pi}_i$ that changes x is an abstract-effect operator. On the other hand, by Restriction 1, the criticality of all effects of α is i, and therefore α is not an abstract-effect operator. Thus, $\overline{\Pi}'_i$ contains α, but does not contain any operator that changes x. But $(x = v)$ does not hold in the initial state, and therefore it does not hold before α in $\overline{\Pi}'_i$. Thus, the preconditions of α are not satisfied before α in $\overline{\Pi}'_i$. Therefore, $\overline{\Pi}'_i$ is not legal, and therefore $\overline{\Pi}_i$ is not a backward semi-ordered refinement of $\overline{\Pi}_j$. □

Since Restriction 2′ imposes less constraints onto an ordered hierarchy than Restriction 2, it allows us to build a finer-grained hierarchy. However, to use this restriction, we need some means to determine whether a precondition of an operator is forbidding. In the next subsection we present algorithms for finding forbidding preconditions of operators.

4.2.3 Forbidding and Non-Forbidding Tests

We consider two kinds of tests for finding forbidding preconditions:

1. *Forbidding test.* Such a test checks conditions that are sufficient but not necessary to guarantee that a precondition is forbidding. In the case of success such a test discovers that some precondition of an operator is *forbidding*, while in the case of failure we do not gain any information.

2. *Non-Forbidding test.* Such a test checks conditions that are sufficient but not necessary to guarantee that a precondition is not forbidding. Such a test may show that some precondition of an operator is *not forbidding*. Again, in the case of failure we gain no information.

First non-forbidding test

We may perform a non-forbidding test using a planning algorithm. To show that a precondition $(x = v)$ of α is not forbidding, we need to find a plan that achieves $S_g = Eff(\alpha)$ starting from an arbitrary initial state in which $(x = v)$ does not hold. We apply a backward-chaining technique to find states from which S_g may be solved. If we find a state S_0 such that $S_0(x) \neq v$ and S_g may be achieved starting from S_0, then $(x = v)$ is not forbidding. If the desired state is not found, we do not gain any information. In general, the backward-chaining is not a polynomial-time algorithm, but we do not need to chain too far. If the algorithm has not succeeded in short time, we may terminate it and apply the polynomial-time tests discussed next.

Second non-forbidding test

In this test we consider only domains *without initial-state axioms*. If $(x_1 = v_1)$ is a precondition of α, and there is an operator α_1 that establishes $(x_1 = v_1)$ and does not negate any precondition of α, then $(x_1 = v_1)$ is *not* a forbidding precondition of α. To see this, we construct a state S_0 in which all preconditions of α hold, except $(x_1 = v_1)$, which does not hold. Then we may apply α_1 to S_0 and receive a new state, $\alpha_1(S_0)$, in which all preconditions of

α hold. We need a domain without initial-state axioms to construct such a state S_0, since in a domain with initial state axioms a required state may be illegal.

Now we present formal description of this test. Let α be an operator with preconditions $(x_1 = v_1), (x_2 = v_2), \ldots, (x_n = v_n)$. Suppose that there exists some operator α_1 that achieves $(x_1 = v_1)$ and does not negate any preconditions of α_1:

(1) $(x_1 = v_1) \in \mathit{Eff}(\alpha_1)$, and
(2) $\forall (x_k = v_k) \in \mathit{Pre}(\alpha)$ either
$x_k \not\sqsubset \mathit{Out}(\alpha_1)$, or
$(x_k = v_k) \in \mathit{Out}(\alpha_1)$

Observe that $(x_1 = v_1) \notin \mathit{Pre}(\alpha_1)$, since the same value of a variable x_1 cannot be both in the preconditions and effects of the same operator. Consider a state S_0 where all preconditions of α_1 hold, all preconditions of α that do not conflict with $\mathit{Pre}(\alpha_1)$ also hold, except for $(x_1 = v_1)$, which does not hold. Formally, S_0 may be described as follows:

let $v_1' \in D(x_1) - \{v_1\}$

$$\text{then } S_0 = \begin{cases} \mathit{Pre}(\alpha_1) \cup \{(x = v) \in \mathit{Pre}(\alpha) \mid x \not\sqsubset \mathit{Pre}(\alpha_1)\} & \text{if } x_1 \sqsubset \mathit{Pre}(\alpha_1) \\ \mathit{Pre}(\alpha_1) \cup \{(x = v) \in \mathit{Pre}(\alpha) \mid x \not\sqsubset \mathit{Pre}(\alpha_1)\} \cup \{(x_1 = v_1')\} & \text{if } x_1 \not\sqsubset \mathit{Pre}(\alpha_1) \end{cases}$$

Observe that x_1 is specified in S_0 in either case, and, since $(x_1 = v_1) \notin \mathit{Pre}(\alpha_1)$, we conclude that $S_0(x_1) \neq v_1$. After applying α_1 to S_0, all preconditions of α hold. Since $(x_1 = v_1)$ does not hold in S_0, this means that $(x_1 = v_1)$ is *not* a forbidding precondition of α.

Thus, we receive a simple non-forbidding test which may be expressed formally as follows:

if $\exists \alpha_1$ such that $(x = v) \in \mathit{Eff}(\alpha_1)$ and $\forall (x_k = v_k) \in \mathit{Pre}(\alpha)$, $(x_k \neq v_k) \notin \mathit{Out}(\alpha_1)$,
then $(x \neq v)$ does not forbid α

This condition may be checked in $\sum_{\alpha \in O}(|\mathit{Pre}(\alpha)| + |\mathit{Eff}(\alpha)|)$ time.

First forbidding test

Let $(x = v)$ be a precondition of α, and no operator in the problem domain achieves $(x = v)$. Then if $(x = v)$ does not hold, it can never be achieved, and α cannot be applied. This gives us a simple forbidding test:

if a precondition $(x = v)$ of α is unachievable,
then it is a forbidding precondition of α

Recall that we store achievable and unachievable values of x separately in the computer memory, and we keep pointers from an operator to each of its effect values. Thus, this test may be performed in $O(1)$ time, just by checking where v is stored.

Second forbidding test

Let α be an operator in a problem domain. We need to find forbidding preconditions of α. We use two colors, *black* and *white*, to paint preconditions of α. After the execution of the algorithm, all *black* preconditions are forbidding. Below we present the description of the algorithm.

Step 1. paint all preconditions of α *black*
Step 2. **for** every *black* precondition $(x = v)$ of α **do**
if there is an operator that achieves $(x = v)$
and does not negate any *black* precondition of α,
then paint $(x = v)$ *white*
Step 3. **if** at least one precondition painted *white* during the execution of Step 2
then go back to Step 2
else end the execution

Termination: The algorithm terminates because at least one *black* precondition is painted *white* during each execution of Step 2, and preconditions never become *black* again.

Correctness: Observe that upon the termination of the algorithm, the preconditions of α are painted in such a way that every operator that achieves some *black* precondition, negates some other *black* precondition. This means that once some *black* precondition does not hold, no sequence of operators may achieve a state where all *black* preconditions hold, and therefore α can never be applied. Thus, all preconditions which are *black* upon the termination of the algorithm are forbidding.

Running time: The algorithm in pseudocode is presented in Table 4.2. The running time of loop 1 is $O(|Pre(\alpha)|)$. Loop 2 is executed at most $(|Pre(\alpha)| + 1)$ times, because during each execution, except the last one, at least one precondition of α is painted *white*. Loop 3 is executed $|Pre(\alpha)|$ times during each execution of loop 2. Lines 9 and 10 are executed $|Out(\alpha_1)|$ times for each α_1 achieving $(x = v)$ during each execution of loop 3. So the maximum number of executions of lines 9 and 10 during one execution of the loop 3 is

$$\sum_{\alpha_1 \in O_{(x=v)}} |Out(\alpha_1)| \leq \sum_{\alpha_1 \in O_{(x=v)}} (|Eff(\alpha)| + |Pre(\alpha)|)$$

(Recall that $O_{(x=v)}$ denotes the set of operators achieving $(x = v)$.) Therefore, the running time of all execution of loop 2 is

$$O(|Pre(\alpha)|^2 \cdot \sum_{\alpha_1 \in O_{(x=v)}} (|Eff(\alpha)| + |Pre(\alpha)|))$$

and the running time of the whole algorithm is the same.

Observe that if a precondition $(x = v)$ of α is not an effect of any operator, the algorithm cannot paint $(x = v)$ *white*. Thus, the algorithm recognizes all unachievable preconditions as forbidding. So this algorithm is stronger than the First Forbidding Test. However, the running time of the Second Forbidding Test is much larger.

While applying the tests described in this section to an operator α, we may learn that some of its preconditions are forbidding, and some others are not. However, there may still be some preconditions $(x_1 = v_1), (x_2 = v_2), \ldots, (x_k = v_k)$ about which we do not know whether they are forbidding or not. A safe way is to assume that they are not forbidding and to impose additional constraints. However, this may reduce the number of levels of the hierarchy. Another way is to show these preconditions to a human expert and ask her advice. The third, better solution of this problem is to assume initially that $(x_1 = v_1), (x_2 = v_2), \ldots, (x_k = v_k)$ are not forbidding, and then use learning technique during planning process to find forbidding preconditions among them. We will describe this approach in Subsection 4.3.1.

Forbidding_Test(O, α)
1. **for** each $(x = v) \in$ *Eff*(α) **do** /* loop 1 */
2. Color$(x = v)$:= *black*;
3. **repeat** /* loop 2 */
4. **for** each $(x = v) \in Pre(\alpha)$ **do** /* loop 3 */
   begin
5. *Make_White* := *False*;
6. **for** each $\alpha_1 \in O$ that achieves $(x = v)$ **do** /* loop 4 */
   begin
7. *Negating* := *False*;
8. **for** each $(x_1 = v_1) \in Out(\alpha_1)$ **do** /* loop 5 */
9. **if** $(x_1 = v_1') \in Pre(\alpha)$ **and** $v_1 \neq v_1'$ **and** Color$(x_1 = v_1)$=*black*
10. **then** *Negating* := *True*;
11. **if not** *Negating*
12. **then** *Make_White* := *True*
   end;
13. **if** *Make_White*
14. **then** Color$(x = v)$:= *white*
   end
15. **until** no precondition is painted *white* during the last execution of the loop;
16. return(all *black* preconditions)

Table 4.2: Finding forbidding preconditions of an operator α

4.2.4 Example

Consider a simple robot world, in which a robot can unlock and open a safe using two keys. The robot is limited in his ability to pick up a key from the floor: when it picks a key, it drops the other key that it holds. To unlock the safe, the robot must hold the both keys, and only an unlocked safe may be opened. If the safe is open, the robot can put the keys into the safe. The problem domain is described with four variables:

- *Key1* and *Key2*, whose values may be *In_Hands*, *On_Floor*, and *In_Safe*
- *Unlocked* and *Open*, whose values may be *True* and *False*

The operators in the problem domain are shown in Table 4.3.

One may verify that the hierarchy based on Restrictions 1 and 2 collapses into a single level. In order to increase the number of levels we observe that, once the robot has dropped one of the keys, it can never hold both of them again. This new information can be obtained by the algorithm *Forbidding_Test*. The algorithm will discover that the operator establishing (*Key1=In_Hands*) negates (*Key2=In_Hands*), and the operator establishing (*Key2=In_Hands*) negates (*Key1=In_Hands*), and therefore both (*Key1=In_Hands*) and (*Key2=In_Hands*) are forbidding preconditions of the operator *unlock*. Therefore Restriction 2′ does not require a variable *Unlocked* to have a higher criticality value than either *Key1* or *Key2*. As a result, we now obtain a three-level abstraction hierarchy shown on Figure 4.1.

operators	preconditions	effects
Pick1	(Key1=On_Floor), (Key2≠In_Hands)	(Key1=In_Hands)
Pick2	(Key2=On_Floor), (Key2≠In_Hands)	(Key2=In_Hands)
Pick1-Drop2	(Key1=On_Floor), (Key2=In_Hands)	(Key1=In_Hands), (Key2=On_Floor)
Pick2-Drop1	(Key2=On_Floor), (Key1=In_Hands)	(Key2=In_Hands), (Key1=On_Floor)
Unlock	(Key1=In_Hands), (Key2=In_Hands)	(Unlocked=True)
Open	(Unlocked=True)	(Open=True)
Put	(Key1=In_Hands), (Key2=In_Hands) (Open=True)	(Key1=In_Safe), (Key2=In_Safe)

Table 4.3: Operators in the robot world

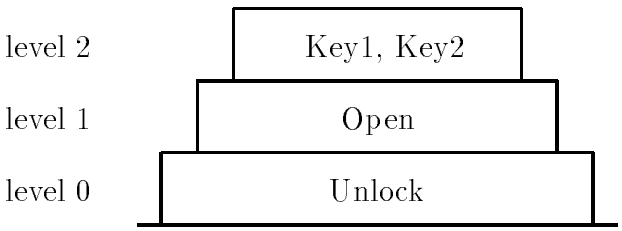


Figure 4.1: Semi-ordered hierarchy for the robot world

4.3 Well-, Greedily, and Perfectly Semi-ordered Hierarchies

In the previous section we discussed backward ordered and semi-ordered hierarchies, whose definitions are based on the backward justified plans. We may use any other kind of justification to define ordered and semi-ordered hierarchies. The definitions are very similar to Definitions 4.3 and 4.6: we just replace the word "backward" with "well-", or "greedily", or "perfectly".

Definition 4.8 (Semi-ordered refinements)
Let (S_0, S_g, Π') *be a correct well- (greedily, perfectly) justified plan at level* $(i+1)$. (S_0, S_g, Π) *is a* well- (greedily, perfectly) semi-ordered refinement *of* (S_0, S_g, Π') *at level* i *if*

(1) (S_0, S_g, Π) *is a refinement of* (S_0, S_g, Π'),
(2) (S_0, S_g, Π) *is well- (greedily, perfectly) justified at level* i, *and*
(3) (S_0, S_g, Π) *remains correct at level* i *upon the removal of all abstract-effect operators*

Notice that a perfectly semi-ordered refinement cannot contain any correct proper subplan, and therefore it cannot contain any abstract-effect operators. This means that every perfectly semi-ordered refinement is also perfectly ordered. Thus, for perfect justification the definition of ordered refinements is identical to the definition of semi-ordered refinements.

	more levels ←—←—		—→—→ less levels
more levels	backward ordered hierarchy		backward semi-ordered hierarchy
↑	well-ordered hierarchy		well-semi-ordered hierarchy
↓	greedily ordered hierarchy		greedily semi-ordered hierarchy
less levels	perfectly ordered hierarchy	=	perfectly semi-ordered hierarchy

Table 4.4: Kinds of ordered and semi-ordered hierarchies

Recall that a backward semi-ordered refinement may be easily converted into a backward ordered refinement (see Lemma 4.1). The same result holds for well- and greedily semi-ordered refinements.

Lemma 4.2 *Let* Π' *be a correct well- (greedily) justified plan at level* $(i+1)$*, and* Π *be a well- (greedily) semi-ordered refinement of* Π' *at the* i*-th level. Let* Π_1 *be obtained from* Π *by removing all abstract-effect operators, and* Π_2 *be a well- (greedily) justified version of* Π_1 *at the* i*-th level. Than* Π_2 *is a well- (greedily) ordered refinement of* Π'*.*

A proof of this lemma is similar to the proof of Lemma 4.1. Lemma 4.2 guarantees that while planning in a semi-ordered hierarchy, we may work with *ordered* (not semi-ordered) refinements.

Definition 4.9 (Well- and greedily semi-ordered hierarchies)
An abstraction hierarchy is well- (greedily) semi-ordered *if any well- (greedily) justified refinement of any correct well- (greedily) justified plan is a well- (greedily) semi-ordered refinement.*

Observe that the backward semi-ordered property of an abstraction hierarchy requires certain conditions to be satisfied for every backward semi-ordered refinement, while the well-semi-ordered property requires these conditions to be satisfied only for well-semi-ordered refinements. Thus if a hierarchy satisfies a backward semi-ordered property, it also satisfies a well-semi-ordered property. In other words, every backward semi-ordered hierarchy is well-semi-ordered. Similarly, every well-semi-ordered hierarchy is greedily semi-ordered, and every greedily semi-ordered hierarchy is perfectly semi-ordered. A similar discussion shows us that every backward ordered hierarchy is well-ordered and so on.

Thus, the backward semi-ordered property of an abstraction hierarchy imposes *more* restrictions onto the criticality values of variables than a well-semi-ordered property. A large number of restrictions reduces our freedom to separate variables into different levels of a hierarchy, and therefore the more restrictions are imposed, the less levels an abstraction hierarchy may have. Thus, a well-semi-ordered hierarchy may have *more* levels of abstraction than a backward semi-ordered hierarchy. Similarly, a greedily semi-ordered hierarchy may have more levels than a well-semi-ordered one and so on. This comparison is summarized in Table 4.4.

In the previous section we have proved that if a planning domain does not have initial-state axioms, Restriction 1 necessarily holds for every backward semi-ordered hierarchy (see

Theorem 4.2). One may check that the same proof works for all kinds of semi-ordered hierarchies.

Theorem 4.3 *If a problem domain does not have initial-state axioms, then any well- (greedily, perfectly) semi-ordered hierarchy satisfies Restriction 1.*

However, Restriction 2′ is *not* a necessary condition of the well-ordered property, as the following example demonstrates.

Example
Assume that you wish to establish a good credit history by making regular payments for a loan. Formally, we have an operator "make regular payments" with a precondition "having loan" and one of the effects "good credit history." Suppose the literal "good credit history" is at the concrete level of an abstraction hierarchy, and "having loan" is at the abstract level. This is a violation of Restriction 2′. However, to receive a loan, you must have a good credit history before you apply for the loan. Thus, in a well-justified plan for achieving good credit history, the operator "make regular payments" is redundant, and we never use this operator at a concrete level of a well-justified plan to achieve "good credit history." Hence, it cannot violate the ordered property.

To describe the same example in a formal fashion, assume that α is an operator with the only precondition $(x = v)$ ("having loan") and the only effect $(x_1 = v_1)$ ("good credit history"). Further assume that for any operator α_1 in the problem domain

- if $(x = v) \in \mathit{Eff}(\alpha_1)$, then $(x_1 = v_1) \in \mathit{Out}(\alpha_1)$, and
- if $(x_1 \neq v_1) \in \mathit{Eff}(\alpha_1)$, then $(x \neq v) \in \mathit{Out}(\alpha_1)$

that is every operator that achieves $(x = v)$, also achieves $(x_1 = v_1)$, and every operators that negates $(x_1 = v_1)$, also negates $(x = v)$. It is easy to see that once some operator α_1 changing x has been applied, $(x = v)$ can hold only when $(x_1 = v_1)$ holds, and therefore no well-justified plan can contain α after α_1.

Thus, the only case when a well-justified plan may contain α is when $(x = v)$ holds in the initial state, and no operator before α changes it. (For example, if you are in debt when you are born, and you are establishing a good credit history by making regular payments for this inherited loan.) But then we may treat $(x = v)$ as the unchangeable operator, and therefore a hierarchy may be semi-ordered even if $\mathit{crit}(x) > \mathit{crit}(x_1)$. An example of such a semi-ordered hierarchy is shown in Table 4.5. □

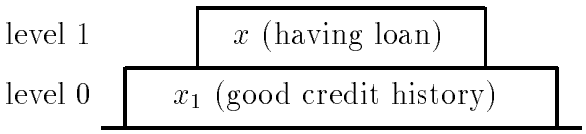
Below we state the modification of Restriction 2′ that allows us to describe sufficient conditions of the well-, greedily, and perfectly ordered properties. The modification is not a single restriction, but four restrictions corresponding to four kinds of justification.

Restrictions 1 and 2″ for backward (well-, greedy, perfect) hierarchy
Let O be the set of operators in a domain. $\forall \alpha \in O, \forall x$ such that $(x = v) \in \mathrm{Pre}(\alpha)$, and $\forall x_1, x_2 \sqsubset \mathrm{Eff}(\alpha)$

1. *$\mathit{crit}(x_1) = \mathit{crit}(x_2)$, and*
2″. *if there exists a correct backward (well-, greedily, perfectly) justified linear plan $(S_0, S_g, \overline{\Pi})$, containing the operator α, such that either*
   - *$(x \neq v) \in S_0$, or*

operator	preconditions	effect
α_1(pay bills regularly)	—	$(x = True)$ (good credit history)
α_2(receive loan)	$(x = True)$ (good credit history)	$(x_1 = True)$ (having loan)
α_3 (regular loan payments)	$(x_1 = True)$ (having loan)	$(x = True)$ (good credit history)

(a) Operators in the problem domain



(b) Semi-ordered hierarchy

Table 4.5: Example of a well-semi-ordered hierarchy, where Restriction 2″ is violated for α_1

- *there exists an operator* α_1 *possibly preceding* α *that achieves* $(x \neq v)$,

then $crit(x) \leq crit(x_1)$

Intuitively Restriction 2″ states that there exists a *justified* plan that contains an operator α, whose precondition $(x = v)$ does not hold at some point before α. Observe that if $(x = v)$ is a forbidding precondition of α, then a plan $\overline{\Pi}$ described in the Restriction 2″ does not exist, and therefore this restriction is weaker than Restriction 2′.

Theorem 4.4 *Restrictions 1 and 2″ for backward (well-, greedy, perfect) hierarchy are sufficient to guarantee the backward (well-, greedily, perfectly) semi-ordered property of an abstraction hierarchy.*

Proof. We present a proof for well-justified plans. Proofs for backward, greedily and perfectly justified plans are similar.

Let us assume that Restrictions 1 and 2″ hold and consider a correct well-justified plan (S_0, S_g, Π') at level $(i+1)$, and its well-justified refinement (S_0, S_g, Π) at level i. Let Π_1 be obtained from Π by removing all abstract-effect operators. We need to show that Π_1 is correct *at the i-th level*. To prove this, we divide the operators of Π_1 into two groups: the operators of Π', and the operators inserted into Π' during refinement process. We need to prove that the operators of the both groups are legal, and that the goal is satisfied in the final state.

Claim 1. Every operator α of Π_1, such that $\alpha \notin \Pi'$, is legal.

Consider an arbitrary precondition $(x = v)$ of α. We need to show that $(x = v)$ holds before α in Π_1. If $crit(\alpha) = i$, then, by Restriction 1, no abstract-effect operator changes $(x = v)$, and therefore $(x = v)$ still holds in Π_1, after the removal of the abstract-effect operators. If $crit(x = v) > i$, then, by Restriction 2″, $(x = v)$ holds in the initial state and no operator possibly preceding α establishes $(x \neq v)$. Therefore, $(x = v)$ holds before α.

Claim 2. Every operator α of Π_1, such that $\alpha \in \Pi'$, is legal, and Π_1 achieves the goal.

The proof of this claim is the same as the proof of the similar claim in Theorem 4.2. □

4.3.1 Learning Technique

Restriction 2″ may seem to be impractical for generating a semi-ordered hierarchy, because we need to examine all possible justified plans in the problem domain in order to impose this restriction. However, we may use a learning technique during the planning process to impose Restriction 2″: if at some point of planning we discover a justified plan that contains some operator α, we impose Restriction 2″ onto α.

Initially, we generate an abstraction hierarchy based on Restriction 1 only. We keep the hierarchy as a directed graph, called a *constraint graph*, and use one of its linearizations for planning. The algorithm for generating this hierarchy is shown in Table 4.6. The constraints are imposed in lines 2–5 of the algorithm. Line 6 finds strongly connected components, which correspond to the levels of the abstraction hierarchy. Observe that Restriction 1 imposes only equality constraints onto criticalities of variables. In terms of edges of the directed graph this means that whenever we draw an edge from x to x_1, we also draw an edge from x_1 to x. Therefore any two vertices of the graph are either in the same component or is not connected by an edge. Thus, there are no edges between strongly connected components, and line 7 imposes an arbitrary linear order onto components. Each component becomes a level of our hierarchy. We are going to keep the graph in a transitively closed form. Notice that after the execution of the algorithm *Impose_Restriction_1*, the graph is transitively closed, since it does not have any edges. (Recall that each connected component is represented as a single vertex.) Imposing Restriction 1 in lines 2–5 takes $\sum_{\alpha \in O} |\mathit{Eff}(\alpha)|$ time. Combining strongly connected components takes $O(|\mathcal{X}|^2)$ time. Linearization process in this algorithm does not take any running time, because there are no edges between strongly connected components, and we may choose an arbitrary order of components. So the total running time of the algorithm is

$$O(\sum_{\alpha \in O} |\mathit{Eff}(\alpha)| + |\mathcal{X}|^2)$$

Since we have not imposed constraints defined by Restriction 2″, it may happen that the resulting hierarchy is not ordered, which may lead to problems during planning. However, when such problems occur, they reveal the lack of constraints, and the necessary additional constraints may be imposed. The maximal number of constraints onto preconditions of α provided by Restriction 2″ is $|\mathit{Pre}(\alpha)|$, and thus the total number of constraints is at most $\sum_{\alpha \in O} |\mathit{Pre}(\alpha)|$. This means that the initial hierarchy will require at most $\sum_{\alpha \in O} |\mathit{Pre}(\alpha)|$ corrections. Thus, we may encounter an inconsistency in the hierarchy at most $\sum_{\alpha \in O} |\mathit{Pre}(\alpha)|$ times during the whole period of use of the hierarchy.

Suppose that during planning at some level of abstraction, we discover a plan with an operator α that does not satisfy Restriction 2″, that is α has a precondition $(x = v)$ such that $(x = v)$ possibly does not hold at some state preceding α, and Restriction 2″ still has not been imposed onto this precondition. Than we impose this constraint and change the abstraction hierarchy accordingly.

The algorithm that searches for such an operator α is presented in Table 4.7. Line 4 of

Impose_Restriction_1
1. *Graph* := create a directed graph where
   (a) every variable in the problem domain is represented as a node
   (b) there are no edges
2. **for** each operator $\alpha \in O$ **do**
   begin
3. choose an arbitrary $x_1 \sqsubset \mathit{Eff}(\alpha)$
4. **for** each $x \sqsubset \mathit{Eff}(\alpha)$ **do**
5. add two edges to *Graph*: from x_1 to x and from x to x_1
   end;
6. *Graph* := Combine_Strongly_Connected_Components(*Graph*);
7. *Hierarchy* := Linearization(*Graph*)

Table 4.6: Imposing Restriction 1 onto the effects of operators

the algorithm checks whether Restriction 2″ has already been imposed onto a precondition $(x = v)$ of an operator α. If it has not, lines 5–10 check whether a precondition $(x = v)$ of an operator α possibly does not hold in some state preceding α. If $(x = v)$ does not hold in the initial state, or if some operator possibly preceding α establishes another value of x, then the algorithm calls the procedure *Add_Constraint*, which imposes Restriction 2″ onto $(x = v)$ and changes the hierarchy accordingly. The execution of lines 3–12 takes $O(|\Pi|)$ time, and since these lines are executed for each precondition of every operator, the total running time of the algorithm is $O(E \cdot |\Pi|)$, not counting the running time of the procedure *Add_Constraint*.

While planning at some level of abstraction, we first find a refinement of the abstract-level plan, then we find a justified version of this refinement, and then we apply the algorithm *Check_Restriction_2″*. If the algorithm does not add any new constraint, we continue to plan on the next lower level of abstraction. If the algorithm finds a missing constraint, the hierarchy is modified, and the planning is started again from the upper level. Observe that the problem of refining the plan is generally NP-hard, the problem to find the justified version of the refinement takes $O(E \cdot |\Pi|^2)$ for backward justification and more for other kinds of justification, and the algorithm *Check_Restriction_2″* takes only $O(E \cdot |\Pi|)$. Thus, the necessity to run this algorithm at each step of planning process does not increase the time complexity of planning.

The algorithm that adds a constraint to the *Graph* and finds a corresponding new hierarchy is shown in Table 4.8. Recall that the running time of linearization and combining strongly connected components is $O(|\mathcal{X}|^2)$ (where $|\mathcal{X}|$ is the number of vertices in the graph), and the average-case running time of the transitive closure algorithm is $O(|\mathcal{X}|^2 \cdot \log\log|\mathcal{X}|)$. Thus, the average-case running time of *Add_Constraints* is $O(|\mathcal{X}|^2 \cdot \log\log|\mathcal{X}|)$. The worst-case running time is $O(|\mathcal{X}|^3)$.

So what is the total cost of our learning method of building a semi-ordered hierarchy? The algorithm *Check_Restriction_2″* is executed at each step of a planning process, but, as we have seen, it does not considerably increase the running time.

Impose_Restriction_1 is executed once and takes $O(\sum_{\alpha \in O} |\mathit{Eff}(\alpha)| + |\mathcal{X}|^2)$ time, and

Check_Restriction_2″(S_0, S_g, Π)
1. **for** each $\alpha \in \Pi$ **do**
2. **for** each $(x = v) \sqsubset \mathit{Pre}(\alpha)$ **do**
   begin
3. choose an arbitrary $x_1 \sqsubset \mathit{Eff}(\alpha)$;
4. **if** there is no edge from x_1 to x in the graph **then**
   begin
5. *Restriction_Holds* := *True*;
6. **if** $(x = v) \notin S_0$
7. **then** *Restriction_Holds* := *False*;
8. **for** each $\alpha_1 \in \Pi$ **do**
9. **if** $\neg(\alpha \prec \alpha_1)$ **and** (α_1 establishes $(x \neq v)$)
10. **then** *Restriction_Holds* := *False*;
11. **if not** *Restriction_Holds*
12. Add_Constraint(x_1, x)
   end
   end

Table 4.7: Searching a justified plan for operators that does not satisfy Restriction 2″

Add_Constraint(x_1, x)
1. add an edge from x_1 to x to *Graph*;
2. *Graph* := Combine_Strongly_Connected_Components(*Graph*);
3. *Graph* := Transitive_Closure(*Graph*);
4. *Hierarchy* := Linearization(*Graph*)

Table 4.8: Adding the edge from x_1 to x to the *Graph* and modifying the hierarchy

Add_Constraint is executed at most $\sum_{\alpha \in O} |Pre(\alpha)|$ times and takes $O(|\mathcal{X}|^2 \cdot \log\log|\mathcal{X}|)$ for each execution. The average-case running time of all executions of these two algorithms is

$$O(\sum_{\alpha \in O} |Eff(\alpha)| + |\mathcal{X}|^2 \cdot \log\log|\mathcal{X}| \cdot \sum_{\alpha \in O} |Pre(\alpha)|)$$

To receive an intuitive idea about the value of this expression, let us consider such a constant *op_size* that, for every operator α, $|Pre(\alpha)| + |Eff(\alpha)| \leq op_size$. Then the total running time of all execution of *Impose_Restriction_1* and *Add_Constraint* is $O(op_size \cdot |\mathcal{X}|^3 \cdot \log\log|\mathcal{X}|)$.

Finally, each time we impose a new constraint, we need to start planning from the beginning, and thus we may need to backtrack to the highest level of the hierarchy because of imposing new constraints at most $\sum_{\alpha \in O} |Pre(\alpha)|$ times.

The learning technique is effective if we often use the same hierarchy. We may use it a lot of times, and we may encounter inconsistency at most $\sum_{\alpha \in O} |Pre(\alpha)|$ times during the whole period of use. It may happen that after a long period of use some necessary constraints are still not inserted, but this means that the lack of these constraints does not create problems for the tasks that our planner usually performs.

4.4 Conclusion

As we have shown, a semi-ordered hierarchy is always finer-grained than the ordered hierarchy produced by ALPINE, which means that a semi-ordered hierarchy contains at least as many levels as an ALPINE-generated one, and that the branching factor during planning in a semi-ordered hierarchy is not higher than during planning in an ordered hierarchy.

We have shown that different definitions of justified plans give rise to different ordered abstraction hierarchies. *More* restrictive kinds of justifications give rise to *less* restrictive conditions for building an abstraction hierarchy, and allow one to generate finer-grained abstraction hierarchies.

While more restrictive justifications lead to abstraction hierarchies with more levels and thus increase the efficiency of planning, the process of finding a more restrictive justification of a given plan takes more time. Thus there is a tradeoff between the number of levels in a semi-ordered abstraction hierarchy and the speed of finding justified versions of plans. Since we need to find justifications of plans at each step of planning process, the necessity to use greedy or perfect justification may considerably slow down the planning process.

We may view Table 4.4 as a spectrum of semi-ordered hierarchies. On one end of the spectrum hierarchies are backward semi-ordered. The running time of finding a backward justified version of a given plan is $O(E \cdot |\Pi|^2)$. On the other end hierarchies are perfectly ordered. Such hierarchies may be finer-grained than any other kind of semi-ordered hierarchies, but the task to find a perfect justification of a given plan is NP-complete, and it may be very time-consuming to find a perfectly justified version of a current plan at each step of planning.

Linear planning is probably most efficient in a greedily semi-ordered hierarchy, because the running time of finding a greedy justification of a linear plan is the same as running time of finding well- and backward ordered justifications.

Chapter 5

Automatically abstracting effects of operators

In this chapter we present another method for increasing the number of levels of an automatically generated abstraction hierarchy by further relaxing Restrictions 1 and 2. We show how these restrictions may be relaxed *without* violating the ordered property for *primary-effect restricted* planning.

Intuitively, we divide the effects of an operator into *primary effects* and *side effects*. We use the operator only for the sake of its primary effects.

For example, suppose you are boiling water in order to prepare tea. The primary effect of this action is obtaining a hot water — this is your main goal. Side effects are spending electricity, heating the air in the room, and maybe burning your fingers. You would not boil water *in order* to spend electricity or *in order* to heat the room.

The planner that uses operators only for the sake of their primary effects is called *primary-effect restricted*. In this chapter we show that primary-effect restricted planners allow us to use abstraction hierarchies with more levels than the hierarchies used by unrestricted planners. Sometimes we are able to build a multi-level ordered hierarchy for a primary-effect restricted planner in cases where the hierarchy based on Restriction 1 and $2'$ collapses into a single level. We present an algorithm that automatically finds primary effects of operators.

5.1 A Motivating Example

Suppose that in the tower of Hanoi example, we add the operators that can move *two disks* at a time, as long as both disks are on the same peg, and there are no disks between them. The operators of this extended tower of Hanoi are listed in Table 5.1.

Both the ordered and semi-ordered hierarchies for this new domain collapse into a single level. Too see this, observe that according to Restriction 1, all effects of operators must have the same criticality. The operator *Move_SM* changes variables *Where_S* and *Where_M*, and therefore $crit(Where_S) = crit(Where_M)$. Similarly, for effects of the operator *Move_ML*, we have $crit(Where_M) = crit(Where_L)$. So the only criticality assignment satisfying Re-

operator	preconditions	effects	primary effects
Move_S(a,b)	(*Where_S*=a)	(*Where_S*=b)	(*Where_S*=b)
Move_M(a,b)	(*Where_M*=a) (*Where_S*≠a), (*Where_S*≠b)	(*Where_M*=b)	(*Where_M*=b)
Move_L(a,b)	(*Where_L*=a) (*Where_S*≠a), (*Where_S*≠b) (*Where_M*≠a), (*Where_M*≠b)	(*Where_L*=b)	(*Where_L*=b)
Move_SM(a,b)	(*Where_S*=a), (*Where_M*=a)	(*Where_S*=b) (*Where_M*=b)	(*Where_M*=b)
Move_SL(a,b)	(*Where_S*=a), (*Where_L*=a) (*Where_M*≠a), (*Where_M*≠b)	(*Where_S*=b) (*Where_L*=b)	(*Where_L*=b)
Move_ML(a,b)	(*Where_M*=a), (*Where_L*=a) (*Where_S*≠a), (*Where_S*≠b)	(*Where_M*=b) (*Where_L*=b)	(*Where_L*=b)

Table 5.1: The operator types in the extended tower of Hanoi domain

striction 1 is

$$crit(Where_S) = crit(Where_M) = crit(Where_L)$$

However, even with the operators for moving two disks, intuitively it is still true that moving the large disk is more difficult than moving the small one. Thus, intuitively one should still consider the movement of a large disk at an abstract level. This example shows a shortcoming of the technique for generating abstraction hierarchies described in the previous chapter: the addition of a few new operators may collapse an abstraction hierarchy into a single level, even though intuition tells us that the abstraction hierarchy should remain intact.

The purpose of the chapter is to remove this deficiency. We achieve this by presenting a new algorithm that constructs ordered abstraction hierarchies based on primary effects of operators.

5.2 Ordered Hierarchies with Primary Effects

A key point to observe in the above example is that, if we want to move the small disk alone, we do not use the operator *Move_SM* or *Move_SL*. It is more natural to move small disk with the operator *Move_S*. Similarly, we use the operator *Move_ML* if we want to move either the large disk, or the large and medium disks together. We do not use *Move_ML* to move the medium disk alone. In other words, an operator is used for the sake of its *primary effects*. In the tower of Hanoi example, we can envision ($Where_L = b$) as the primary effect of $Move_SL(a, b)$ operator, and ($Where_S = b$) as its *side effect*. The set of primary effects of the tower of Hanoi operators are listed in Table 5.1.

The purpose of recognizing primary effects is to restrict the complexity of a problem-solving process by reducing the branching factor of the search space. When achieving the value of some variable, a planner needs to consider only operators whose primary effects

match the variable. If a planner uses operators only for the sake of their primary effects, it is said to be *primary-effect restricted*. Unfortunately, we cannot give the universal precise definition of the primary-effect restriction, because it depends on a behavior of a particular planning systems. So to explain the meaning of the primary-effect restriction, we briefly describe common planning algorithms.

A formal overview of several main types of planning algorithms may be found in [Minton *et al.*, 1991]. Here we present only short informal description of both linear and nonlinear planners.

Linear planners.
Linear planning algorithms such as GPS [Newell and H. Simon, 1972] and STRIPS [Fikes and Nilsson, 1971] build plans by backward chaining from the goal state. The preconditions of some operators on intermediate stages of plan generation are not satisfied, and the planner tries to achieve them by inserting new operators. For example, consider the tower of Hanoi domain with two disks, small and large, and the planning problem with

Initial state: $(Where_S = 1), (Where_L = 1)$
Goal state: $(Where_L = 2)$

(see Figure 5.1). The plan

$$\longrightarrow Move_L(1,2) \longrightarrow$$

achieves the goal, but the precondition $(Where_S \neq 1)$ of the operator $Move_L(1,2)$ is not satisfied. So the planner inserts an operator that satisfies this precondition and receives a correct plan

$$\longrightarrow Move_S(1,3) \longrightarrow Move_L(1,2) \longrightarrow$$

Nonlinear planners.
Nonlinear planners such as NOAH [Sacerdoti, 1975] and TWEAK [Chapman, 1987] achieve preconditions of operators not only by inserting new operators, but also by imposing the order constraints onto existing operators. For example, consider the problem shown in Figure 5.1 with the same initial state and

Goal state: $(Where_S = 3), (Where_L = 2)$

The plan

$$\longrightarrow \begin{array}{c} \nearrow \\ \searrow \end{array} \begin{array}{c} Move_S(1,3) \\ Move_L(1,2) \end{array} \begin{array}{c} \searrow \\ \nearrow \end{array} \longrightarrow$$

achieves the goal, but the precondition $(Where_S \neq 1)$ of $Move_L(1,2)$ may not be satisfied if this operator is executed first. So a nonlinear planner achieves this precondition by imposing the time-precedence constraint "$Move_S$ before $Move_L$", and receives a correct plan

$$\longrightarrow Move_S(1,3) \longrightarrow Move_L(1,2) \longrightarrow$$

(a) Initial state (b) Goal state

Figure 5.1: Two-disk tower of Hanoi problem

A planning algorithm is primary-effect restricted if, when inserting a new operator to achieve some precondition $(x = v)$ of an operator α, it always inserts an operator with the *primary* effect $(x = v)$, and does not use an operator that achieves $(x = v)$ as its side effect. However, while imposing time-precedence constraints onto existing operators, the algorithm may *not* obey primary-effect restriction, and may impose such a constraint that the preconditions of α are satisfied by side effects of some operator.

The definition of primary-effect restricted planning, unlike all other definitions in this thesis, depends on the planning algorithm. All definitions and all theorems presented in the previous chapters are independent of the behavior of a planning algorithm, as long as the planner produces correct plans and uses a justification procedure when appropriate. However, the sufficient condition of the ordered property presented in this section depends on the behavior of planning algorithm: this condition holds only for primary-effect restricted planners.

In the next section we present an algorithm for *automatically* finding the primary effects of operators. But for now, we assume that the primary effects of the operators in a domain have been found. The set of primary effects of an operator α is denoted by *Prim-Eff*(α_1), and the set of side effects by *Side-Eff*(α_1).

Now we describe construction of a finer-grained ordered abstraction hierarchy based on the primary effects, for use by a primary-effect restricted planner. Such a hierarchy is called *primary-effect restricted*. Consider the following modified ordered restriction:

Restrictions 1a and 2a *Let O be the set of operators in a domain. $\forall \alpha \in O$, $\forall x$ such that $(x = v) \in$ Pre(α), $\forall x' \sqsubset$ Eff(α), and $\forall x_1, x_2 \sqsubset$ Prim-Eff(α)*

1a. $crit(x') \leq crit(x_1) = crit(x_2)$, and

2a. if $(x = v)$ is achievable, then $crit(x) \leq crit(x_1)$

This restriction formalizes the syntactic conditions behind the algorithm used by ALPINE ([Knoblock, 1991a], page 83). Stated simply, the criticality values of all *primary* effects of an operator are the same, and they are no less than the criticalities of secondary effects and achievable preconditions of the operator. For example, one can verify that the abstraction hierarchy shown in Figure 5.2 for the extended tower of Hanoi domain with primary effect as indicated in Table 5.1 satisfies Restrictions 1a and 2a.

For a primary-effect restricted planner, Restrictions 1a and 2a provide a sufficient condition of the backward ordered property.

Theorem 5.1 *If an abstraction hierarchy used by a primary-effect restricted planner satisfies Restrictions 1a and 2a, the hierarchy is ordered.*

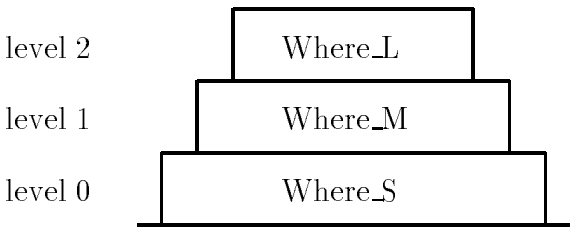


Figure 5.2: Semi-ordered hierarchy for the extended tower of Hanoi

Proof. Consider some correct plan (S_0, S_g, Π') at the $(i+1)$-th level of abstraction, and the process of refining Π' at the i-th level. To prove that the final refinement Π of Π' is ordered, we need to show that no abstract-effect operator is inserted during the refinement process. We divide the refinement process into finite number of steps such that at each step the planning algorithm either inserts a new operator or imposes a time-precedence constraint. We denote the intermediate plan obtained after the k-th step by Π_k, and the set of *achievable* preconditions of operators and goal values that does not hold in Π_k by P_k. We use induction on k to prove that no operator of Π_k inserted during the refinement process has any effect with the criticality higher than i.

Inductive hypothesis:
(1) for any $\alpha \in \Pi_k$ such that $\alpha \notin \Pi'$, for any $x_1 \sqsubset \mathit{Eff}(\alpha)$, $\mathit{crit}(x_1) \leq i$, and
(2) for any precondition $(x = v) \in P_k$, $\mathit{crit}(x) \leq i$

Base: For $k = 0$, that is before any changes of the abstract plan, $\Pi_0 = \Pi'$. Hypothesis 1 trivially holds, since no operators have been inserted into Π'. Hypothesis 2 holds because Π' is correct at the $(i+1)$-th level of abstraction, and therefore all preconditions with criticalities higher than i are satisfied.

Step: Assume that Hypotheses 1 and 2 hold after the execution of the k-th step, and consider the $(k+1)$-th step of refining. Clearly, imposing a new time-precedence constraint cannot violate the inductive hypothesis, and so assume that at the $(k+1)$-th step the planner inserts a new operator α that achieves one of the unsatisfied preconditions $(x = v) \in P_k$. By Hypothesis 2, $\mathit{crit}(x) \leq i$. Since the planner is primary-effect restricted, $(x = v)$ is a primary effect of α, and by Restriction 1a all effects of α have criticalities no larger than i. Therefore, Hypothesis 1 still holds after the insertion of α, in the resulting plan Π_{k+1}. Observe that since all effects of α have criticalities no larger than i, its insertion cannot violate any higher-level precondition of any operator. Also, by Restriction 2a, all achievable preconditions of α have criticalities no larger than i. Therefore, no unsatisfied achievable higher-level preconditions may appear in the plan after the insertion of α, and thus P_{k+1} cannot contain any preconditions whose criticalities are larger than i. Therefore, Hypothesis 2 also holds after the $(k+1)$-th step. □

Unfortunately, we cannot use the notion of semi-ordered hierarchies and relax Restriction 2a the way we have relaxed Restriction 2 (see Theorem 4.2) for an arbitrary primary-

effect restricted planner. The proof of Theorem 4.2 does not work for primary-effect restricted hierarchies, because it crucially depends on the equality of the criticalities of all effects of an operator (see Claim 1, Case 1 in the proof of the theorem). However, Restriction 2a *may* be relaxed if we impose an additional restriction onto a planner: we demand our planner to be not only primary-effect restricted, but also *forbidding-restricted.*

Restrictions 1a and 2a′ *Let O be the set of operators in a domain. $\forall \alpha \in O$, $\forall x$ such that $(x = v) \in \text{Pre}(\alpha)$, $\forall x' \sqsubset \text{Eff}(\alpha)$, and $\forall x_1, x_2 \sqsubset \text{Prim-Eff}(\alpha)$*

1a. $crit(x') \leq crit(x_1) = crit(x_2)$, and

2a′. if $(x = v)$ is not a forbidding precondition, then $\text{crit}(x) \leq \text{crit}(x_1)$

Theorem 5.2 *An abstraction hierarchy for a primary-effect restricted forbidding-restricted planner is ordered if and only if it satisfies Restrictions 1a and 2a′.*

A formal proof of this theorem is the same as the proof of Theorem 5.1, with P_k being the set of *non-forbidding* unsatisfied preconditions after the k-th step of the refinement process.

Informally, the possibility of replacing Restriction 2a with less restrictive 2a′ may be explained as follows. Let $(x_1 = v_1)$ be a primary effect of some operator α, where the criticality of x is i, and $(x = v)$ be a forbidding precondition of α. Suppose that while refining some abstract-level plan at level i, the planner applies α to achieve $(x_1 = v_1)$. Since the planner is primary-effect restricted, it will not then insert an operator that achieves the precondition $(x = v)$ of α, and thus it will not violate any higher-level literal even if the criticality of $(x = v)$ is higher than i.

5.3 Automatically Finding Primary Effects

There are different ways of finding primary effects of operators, which can be grouped into the following three alternatives:

1. **All Effects Are Primary Effects.** This option is implicitly used in Restrictions 1 and 2′ described in the previous chapter. It is also used as default by Knoblock's ALPINE [Knoblock, 1991a] if no primary effects are provided by a user. As we have demonstrated, it may lead to generating a hierarchy with too small number of abstraction levels.

2. **User-defined Primary Effects.** This is the approach taken by ALPINE and many other systems. For example, the ABTWEAK [Yang *et al.*, 1991] system depends on the user to define the set of primary effects of operators.

3. **Automatically Selecting Primary Effects.** This is the approach we are taking. The advantage of our approach is that we could now feel free to select primary effects in such a way as to *maximize* the total number of abstraction levels.

Now we describe an algorithm for automatically selecting primary effects of operators. The goal of the algorithm is to maximize the number of abstraction levels.

To ensure completeness of abstract planning, we would like every achievable value of each variable to be a primary effect of some operator. If a value is not a primary effect of any

Choose_Primary_Effects
1. *Graph* := Create a directed graph where
   (a) every literal in the problem domain is represented as a node
   (b) there are no edges
2. User-Defined_Primary_Effects;
3. **for** $m := 1$ to n **do**
   begin
4. **for** each operator α that achieves values of m distinct variables **do**
5. **if** *User-defined-Prim-Eff*(α) is empty
        /* human user have not defined primary effects of α */
6. **then** Choose_Primary_Effect_Of_Operator(α);
7. **for** each value $(x = v)$ that is achieved by m distinct operators **do**
8. **if** $(x = v)$ still is not chosen as a primary effect of some operator
9. **then** Make_Value_Be_Primary_Effect($x = v$)
   end;
10. Choose_Primary_Effects_According_To_Graph;
11. *Hierarchy* := Linearization(*Graph*)

Table 5.2: Creating an ordered hierarchy

operator, then it cannot be achieved. Likewise, every operator must have a primary effect. If an operator has no primary effects, it can never be used in planning.

Our algorithm *Choose_Primary_Effects* is shown in Table 5.2. Its input is a set of operators in a domain, and its output is a selection of primary effects for each operator. The algorithm chooses primary effects by building a constraint graph of variables. The variables of a problem domain are represented as nodes of a directed graph, and constraints are represented as directed edges. An edge from x_1 to x_2 indicates that $crit(x_1) \geq crit(x_2)$. Strongly connected components of the graph correspond to the levels of abstraction. At first the graph does not have any edges. While choosing primary effects, we try to maximize the number of strongly connected components. In the code of the algorithm, the notation $|Graph|$ denotes the number of strongly connected components of the *Graph*.

The algorithm allows the user to select primary effects of some operators, and then chooses primary effects for the other operators. The procedure *User-defined_Primary_Effects*, presented in Table 5.3, imposes constraints described by Restrictions 1a and 2a onto user-defined primary effects.

The algorithm starts to select primary effects of the remaining operators by considering operators each of which establishes a value of exactly one variable. Recall, that we wish every operator to have at least one primary effect. So for each operator α that has a unique effect $(x = v)$, we make $(x = v)$ the primary effect of α, and add directed edges from x to all precondition variables of α. Then we consider each value achieved by a unique operator. Since each achievable value must be a primary effect of some operator, we make every value achieved by a unique operator a primary effect of the corresponding operator. At the second step, we consider the set of operators that establish values of two distinct variables, and the values that are achieved by two different operators, and so on.

User-defined_Primary_Effects
1a. **for** each operator α **do**
2a. **if** *User-defined-Prim-Eff*(α) is not empty
then begin
3a. *Prim-Eff*(α) := *User-defined-Prim-Eff*(α);
4a. choose an arbitrary $x_1 \sqsubset$ *Prim-Eff*(α);
5a. **for** each $x_2 \sqsubset$ *Prim-Eff*(α) **do**
6a. add an edge from x_2 to x_1 to *Graph*;
7a. **for** each $x' \sqsubset$ *Eff*(α) **do**
8a. add an edge from x_1 to x' to *Graph*;
9a. **for** each $(x = v) \in$ *Pre*(α) **do**
10a. **if** $(x = v)$ is achievable
11a. **then** add an edge from x_1 to x to *Graph*;
end;
12a. Combine_Strongly_Connected_Components(*Graph*)

Table 5.3: Imposing constraints onto user-defined primary effects

At the m-th step, we perform two operations:

1. choose primary effects of each operator that establishes values of m different variables, and

2. for every value $(x = v)$ that is achieved by m different operators, make $(x = v)$ a primary effect of one of the corresponding operators

Each time when we select a primary effect, we choose it such a way as to maximize the number of strongly connected components. In other words, we use a greedy strategy by making locally optimal choices at each step. The total number of steps performed by the algorithm, n, is such that

- no operator has more than n effects, and
- no value $(x = v)$ is established by more than n distinct operators

Observe that the fewer effects the operator has, the earlier we select its primary effects. The intuition behind this order is as follows. When we consider single-effect operators, the choice of the primary effect is uniquely determined, so a wrong choice cannot be made. When we consider 2-effect operators, we choose a primary effect out of 2 effects, and the probability of the wrong choice is not very large. Generally, the more effects an operator has, the larger the probability to make a wrong choice is. However, if some constraints have been imposed before we choose a primary effects of an operator, there are some chances that a primary effect is already implicitly selected by previously imposed constraints, and thus we do not need to make any choice and cannot be wrong. Even if a primary effect still is not determined by previously imposed constraints, these constraints give us an additional information and decrease the probability of a wrong choice. That is why we wish to consider operators with a lot of primary effects, where the probability of a wrong choice is high, after imposing constraints onto operators with fewer effects.

```
Choose_Primary_Effect_Of_Operator(α)
/* the algorithm uses temporary variables Graph1 and Graph2 */
1b.  Max_Number_Of_Components := 0;
2b.  for each (x = v) ∈ Eff(α_1) do
     begin
3b.     Graph1 := Graph;
4b.     for each (x_1 = v_1) ∈ Eff(α) do
5b.        add an edge from x to x_1 in Graph1;
6b.     for each (x_2 = v_2) ∈ Pre(α) do
7b.        if (x_2 = v_2) is achievable
8b.           then add an edge from x to x_2 to Graph1;
9b.     Combine_Strongly_Connected_Components(Graph1);
10b.    if |Graph1| > Max_Number_Of_Components;
           then begin
11b.                Primary_Effect := (x = v);
12b.                Graph2 := Graph1;
13b.                Max_Number_Of_Components := |Graph1|
                 end
     end;
14b. Graph := Transitive_Closure(Graph2);
15b. Prim-Eff(α) := {(x = v)}
```

Table 5.4: Selecting a primary effect of an operator α

We do not have any theoretical evidence to support this intuition. Experiments in simple problem domains such as the tower of Hanoi, blocks worlds, and simple robot's worlds have shown that our algorithm usually produces the optimal or near-optimal hierarchy. However, we did only a few experiments in very simple domains, and the evidence is not enough to confirm general efficiency of the algorithm.

Selecting a primary effect of every operator α is performed by the algorithm *Choose_Primary_Effects_Of_Operator* (see Table 5.4). Let $(x_1 = v_1), (x_2 = v_2), \ldots, (x_m = v_m)$ be effects of α, and *Graph* be the constraint graph before a primary effect of α is selected. First the algorithm tries to make $(x_1 = v_1)$ a primary effect of an operator by adding constraints defined by Restrictions 1a and 2a$'$: the algorithm adds directed edges from x_1 to all other effect variables of α and to all precondition variables of α. After adding these edges, the algorithm receives some new graph $Graph_1$. Then the algorithm tries to make $(x_2 = v_2)$ a primary effect of α, and receives a new graph $Graph_2$. Similarly, it receives graphs $Graph_3, \ldots, Graph_m$. The algorithm counts the number of strongly connected components in each of the graphs, and chooses graph $Graph_k$ with the largest number of components. Since the purpose is to maximize the number of strongly connected components, $(x_k = v_k)$ is finally chosen as a primary effect of α. Observe, that we keep the resulting graph in the transitively closed form (see Line 14). This allows us to compare the criticalities of two variables in constant time.

```
Make_Value_Be_Primary_Effect(x = v)
/* the algorithm uses temporary variables Graph1 and Graph2 */
1c.  Max_Number_Of_Components := 0;
2c.  for each α that achieves (x = v) do
     begin
3c.      Graph1 := Graph;
4c.      for each (x1 = v1) ∈ Eff(α1) do
5c.          add an edge from x to x1 to Graph1;
6c.      for each (x2 = v2) ∈ Pre(α1) do
7c.          if (x2 = v2) is achievable
8c.              then add an edge from x to x2 to Graph1;
9c.      Combine_Strongly_Connected_Components(Graph1);
10c.     if |Graph1| > Max_Number_Of_Components;
             then begin
11c.                 Chosen_Operator := α;
12c.                 Graph2 := Graph1;
13c.                 Max_Number_Of_Components := |Graph1|
                 end
     end;
14c. Graph := Transitive_Closure(Graph2);
15c. Prim-Eff(Chosen_Operator) := Prim-Eff(Chosen_Operator) ∪ {(x = v)}
```

Table 5.5: The algorithm makes $(x = v)$ to be a primary effect of some operator

The algorithm *Make_Value_Be_Primary_Effect* accepts some value $(x = v)$, considers all operators that achieve $(x = v)$ and makes $(x = v)$ a primary effect of one of them. The algorithm is presented in Table 5.5. This algorithm is very similar to the algorithm *Choose_Primary_Effect_Of_Operator*. Let α_1, α_2,...,α_m be operators that achieve $(x = v)$. The algorithm tries to make $(x = v)$ a primary effect of α_1 by imposing the corresponding constraints, and determines the number of strongly connected components in the resulting graph. Then the algorithm tries to make $(x = v)$ a primary effect of α_2, then of α_3, and so on till α_m, and chooses the case in which the resulting graph has the largest number of strongly connected components.

Observe that while selecting primary effects, we do not store the selected primary effects explicitly. Some primary effects have been explicitly stored in the set *Prim-Eff* in lines 15b and 15c, but these are only *some*, not all primary effects. Instead, the primary effects are determined implicitly by constraints imposed onto the graph: if the criticality of an effect of an operator is not less than the criticalities of all other effects, and not less than the criticalities of all achievable preconditions, then the effect is primary. In terms of graph edges this means that the effect is primary if there are direct arrows from the effect to all other effects and to all achievable preconditions of the operator. After the constraint graph is completely built, we wish to find the primary effects of each operator α and store them explicitly as a set *Prim-Eff*(α). This is performed by the algorithm *Choose_Primary_Effects_According_To_Graph*, presented in Table 5.6. The algorithm chooses

Choose_Primary_Effects_According_To_Graph
1d. **for** each operator $\alpha \in O$ **do**
    begin
2d. let $(x = v)$ be the primary effect of α chosen in line 4a, 15b, or 15c;
3d. **for** each $(x_1 = v_1) \in \mathit{Eff}(\alpha)$ **do**
4d. **if** x and x_1 are in the same connected component of *Graph*
        /* that is $\mathit{crit}(x) = \mathit{crit}(x_1)$ */
5d. **then** $\mathit{Prim\text{-}Eff}(\alpha) := \mathit{Prim\text{-}Eff}(\alpha) \cup \{(x_1 = v_1)\}$
    end

Table 5.6: The algorithm stores the primary effects of each operator α in the set $\mathit{Prim\text{-}Eff}(\alpha)$

the primary effects of operators according to the imposed constraints: for each operator α it compares the criticality of every effect of α with the primary effect found by the procedure *Choose_Primary_Effect_Of_Operator*. All effects of α whose criticalities are equal to the criticality value of the primary effect, are added to the set of primary effects of α.

Now let us find the time complexity of the algorithm. We start with the time complexity of *Choose_Primary_Effects_Of_Operator*. Recall that the vertices of the *Graph* are variables of the problem domain, and therefore the number of vertices before combining strongly connected components is $|\mathcal{X}|$. The amount of memory necessary for storing the *Graph* is $O(|\mathcal{X}|^2)$, and therefore the running time of line 3b is also $O(|\mathcal{X}|^2)$. The running time of the loop in lines 4b–5b is $O(|\mathit{Eff}(\alpha)|)$, and the running time of the loop in lines 6b–8b is $O(|\mathit{Pre}(\alpha)|)$. Combining strongly connected components in line 9a takes $O(|\mathcal{X}|^2)$ time. Counting the number of vertices of *Graph*1 in lines 10b and 13b takes $O(|\mathcal{X}|)$ time, and line 11b takes $O(|\mathcal{X}|^2)$ time. Thus, the total running time of lines 3b–13b is

$$O(|\mathit{Eff}(\alpha)| + |\mathit{Pre}(\alpha)| + |\mathcal{X}|^2)$$

Since these lines are executed for each effect of α, the running time of all executions of the loop in lines 2b–13b is

$$O(|\mathit{Eff}(\alpha)| \cdot (|\mathit{Eff}(\alpha)| + |\mathit{Pre}(\alpha)| + |\mathcal{X}|^2))$$

Line 14b takes $O(|\mathcal{X}|^2 \cdot \log\log|\mathcal{X}|)$ in the average case, so the average-case running time of the algorithm *Choose_Primary_Effects_Of_Operator* for an operator α is

$$O(|\mathit{Eff}(\alpha)| \cdot (|\mathit{Eff}(\alpha)| + |\mathit{Pre}(\alpha)| + |\mathcal{X}|^2) + |\mathcal{X}|^2 \cdot \log\log|\mathcal{X}|)$$

Observe that for each variable x, at most one value of x is specified in the set of preconditions of α, and therefore $|\mathit{Pre}(\alpha)| \leq |\mathcal{X}| \leq |\mathcal{X}|^2$, and similarly $|\mathit{Eff}(\alpha)| \leq |\mathcal{X}|^2$. Therefore, the above running time expression may be simplified to

$$O(|\mathit{Eff}(\alpha)| \cdot |\mathcal{X}|^2 + |\mathcal{X}|^2 \cdot \log\log|\mathcal{X}|)$$

Since we call this algorithm exactly once for each operator in the problem domain, the average-case running time of all calls of the algorithm is

$$O(\sum_{\alpha \in O}(|\mathit{Eff}(\alpha)| \cdot |\mathcal{X}|^2 + |\mathcal{X}|^2 \cdot \log\log|\mathcal{X}|)$$

which may be rewritten as

$$O(|\mathcal{X}|^2 \cdot \sum_{\alpha \in O} |\mathit{Eff}(\alpha)| + |O| \cdot |\mathcal{X}|^2 \cdot \log\log|\mathcal{X}|) \qquad (1)$$

A similar derivation shows that the average-case running time of all executions of the algorithm *Make_Value_Be_Primary_Effect* is

$$O(\sum_{x \in \mathcal{X}} \sum_{v \in D(x)} \sum_{\alpha \in O_{(x=v)}} (|\mathit{Eff}(\alpha)| \cdot |\mathcal{X}|^2 + |\mathcal{X}|^2 \cdot \log\log|\mathcal{X}|)) \qquad (2)$$

Observe that the expression

$$\sum_{x \in \mathcal{X}} \sum_{v \in D(x)} \sum_{\alpha \in O_{(x=v)}} (\ldots)$$

means "for each variable x, for each value $(x = v)$, consider once every operator that establishes $(x = v)$", which may be restated as "for each operator consider once each of its effects $(x = v)$", and thus the original sum expression may be replaced with

$$\sum_{\alpha \in O} \sum_{(x=v) \in \mathit{Eff}(\alpha)} (\ldots)$$

After substituting this sum expression into (2), it is easy to verify that expressions (1) and (2) are equivalent, and thus the running time of all executions of *Make_Value_Be_Primary_Effect* equals (1).

One execution of the main loop of the procedure *User-defined_Primary_Effects* (lines 1a–11a) takes $O(|\mathit{Pre}(\alpha)| + |\mathit{Eff}(\alpha)|)$ time, and therefore the time of all executions of the loop is

$$O(\sum_{\alpha \in O} (|\mathit{Pre}(\alpha)| + |\mathit{Eff}(\alpha)|))$$

The running time of the line 12a is $O(|\mathcal{X}|^2)$, and thus the overall running time of *User-Defined_Primary_Effects* is

$$O(\sum_{\alpha \in O} (|\mathit{Pre}(\alpha)| + |\mathit{Eff}(\alpha)|) + |\mathcal{X}|^2)$$

which is clearly less than (1).

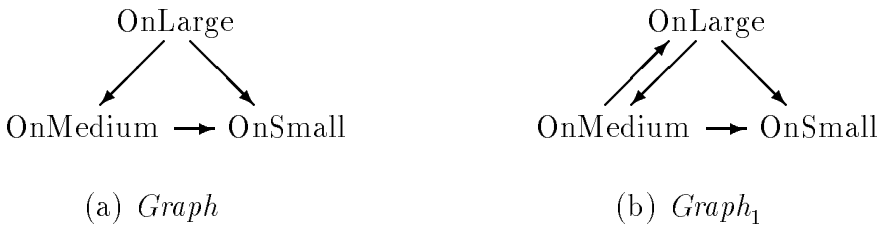
Finally, the running time of the algorithm *Choose_Primary_Effects_According_To_Graph* is

$$O(\sum_{\alpha \in O} |\mathit{Eff}(\alpha)|)$$

and the running time of *Linearization* in line 11 is $O(|\mathcal{X}|^2)$, so the running time of these two algorithms is also less than (1). Therefore, the running time of the whole algorithm *Choose_Primary_Effects* is described by expression (1).

Example

We consider the extended tower of Hanoi domain. Each of the operators *Move_S*, *Move_M*, *Move_L* achieves one value, and at the first step the algorithm makes this value a primary



(a) *Graph* (b) $Graph_1$

Figure 5.3: Graphs in the extended tower of Hanoi example

effect. After performing this step, the *Graph* is as shown on Figure 5.3a. Then the algorithm considers operators that achieve two distinct literals. These operators are *Move_ML*, *Move_SM*, and *Move_SL*.

The effect variables of *Move_ML* are *Where_M* and *Where_L*. One of them must be chosen as a primary effect. If *Where_L* is a primary effect, its criticality must be at least as great as the criticality of *Where_M* and the criticalities of all preconditions of *Move_ML*. These restrictions already hold in the *Graph*, so it is not necessary to add new restrictions. If *Where_M* is chosen as a primary effect of *Move_ML*, we must have

$$crit(Where_M) \geq crit(Where_L)$$

After the constraint defined by this inequality is added to the *Graph*, we receive a new graph $Graph_1$ shown in Figure 5.3b. $Graph_1$ contains fewer strongly connected components than *Graph*. Since the purpose is to maximize the number of strongly connected components, the algorithm finally chooses *Where_L* to be a primary effect of *Move_ML*.

Then the algorithm uses the same method to choose primary effects of *Move_SM* and *Move_SL*. One may check that the algorithm chooses *Where_M* to be a primary effect of *Move_SM*, and *Where_L* to be a primary effect of *Move_SL*. After linearizing the resulting graph, the algorithm receives the hierarchy shown in Figure 5.2. □

5.4 Advantages and Limitations of using Primary Effects

In this section we discuss the advantages and limitations of an abstraction hierarchy based on Restrictions 1a and 2a, as compared to hierarchies described in the previous section. We compare the two types of abstraction levels in terms of the number of hierarchies generated by each algorithm. Also, we discuss the completeness of the resulting planning system.

Advantages

1. Restrictions 1a and 2a are less restrictive than 1 and 2, and Restriction 2a′ is less restrictive than 2′. Thus, primary-effect restricted hierarchies based on this restrictions have more levels of abstractions than usual backward semi-ordered hierarchies.

2. If we need to establish some value $(x = v)$ during the planning process, we may use only operators with a *primary* effect $(x = v)$, not all the operators that achieve $(x = v)$. This

reduces the branching factor of the search.

Now we discuss the *completeness* of a planner that uses an abstraction hierarchy produced by our algorithm. We wish to ensure that if a planning problem is solvable, then a search tree expanded by a primary-effect restricted planner contains a solution of this problem. That is, for any initial state S_0 and goal state S_g, if there is a plan Π that achieves S_g starting from S_0, then there is some plan (not necessarily Π) in the search space expanded by a planner that also achieves S_g from the initial state S_0. If a planner uses a breadth-first search, then this property guarantees that, given enough time, the planner will find a solution of any solvable problem.

We assume that while refining any abstract level plan Π at any lower level i, a planner is able to find every perfectly justified refinement of Π. In other words, for any abstraction level i and any $(i+1)$-level plan Π, the search space expanded by a planner within the i-th level of abstraction while refining Π contains all perfectly justified refinements of Π. Most planners, such as ABSTRIPS, TWEAK, and NONLIN, satisfy this assumption.

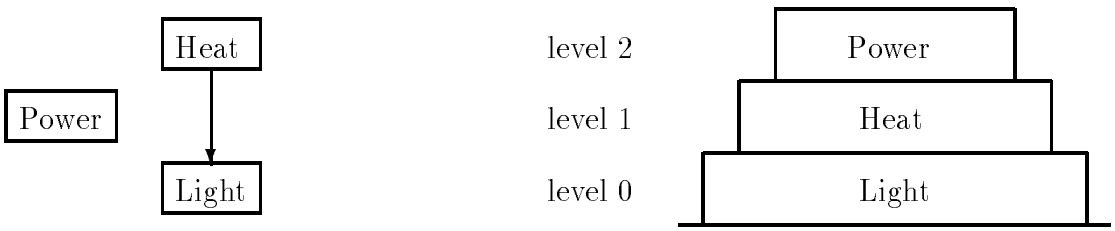
To make sure that completeness is not lost while planning in a hierarchy of abstraction spaces, we have to guarantee that if there is a plan that achieves S_g starting from S_0, then a planner may find it while using the abstraction hierarchy. This implies that there exists a sequence $\Pi_0, \Pi_1, \ldots, \Pi_{n-1}$ of plans such that Π_{n-1} is a perfectly justified plan that solves the problem at level $(n-1)$, Π_{n-2} is a perfectly justified ordered refinement of Π_{n-1} that solves the problem at level $(n-2)$, and so on until Π_0, which solves the problem at the lowest level of abstraction. If a hierarchy satisfies this requirement, we say that it has the *completeness property.*

Definition 5.1 (Completeness property)
An abstraction hierarchy with n levels of abstraction is said to have the completeness property *if, whenever a goal state S_g may be achieved from an initial state S_0 at the lowest level of abstraction, there exists a sequence $\Pi_0, \Pi_1, \ldots, \Pi_{n-1}$ of plans such that*

- *for every $i \in [0..(n-1)]$, (S_0, S_g, Π_i) is a correct perfectly justified plan at the i-th level of abstraction, and*
- *for every $i \in [0..(n-2)]$, Π_i is an ordered refinement of Π_{i+1}*

Yang and Tenenberg defined the *monotonicity property* [Yang *et al.*, 1991], which is similar to our completeness property. The only difference between the two definitions is that in the definition of the monotonicity property, a refinement Π_i of a higher-level plan Π_{i+1} may *not* be an *ordered* refinement. Since every refinement of every plan in an ordered hierarchy is an ordered refinement, the monotonicity property and completeness property are equivalent in the case of an ordered hierarchy. Yang and Tenenberg have shown that every abstraction hierarchy has the monotonicity property [Yang *et al.*, 1991], and thus guarantees completeness of planning.

However, the completeness property and monotonicity property are *not* equivalent for a primary-effect restricted hierarchy. While every hierarchy has the monotonicity property, a primary-effect restricted hierarchy may *not* have a completeness property, and thus primary-effect restricted planning may not be complete.



(a) Constraint graph (b) Abstraction hierarchy

Figure 5.4: Constraint graph and an abstraction hierarchy for the Fireplace domain

A few experiments that we made have shown that the conditions used in the algorithm *Choose_Primary_Effects*, which are

(1) every operator has a primary effect, and

(2) every achievable value of each variable is a primary effect of some operator,

usually lead to the generation of a hierarchy with the completeness property, but again the experiments were made only with several simple domains such as variations of the tower of Hanoi and blocks world, and they are not sufficient to draw a conclusion about general efficiency of the algorithm. Below we present an example of a hierarchy that satisfies conditions (1) and (2), but does not satisfy the completeness property.

Example (Hierarchy without the completeness property)
Consider the operator "set fire in a fireplace" with the effects "the room is heated" and "the room is lighted", and the operator "turn electric light on" with the precondition "electric power is available" and the effect "the room is lighted" (see Table 5.7). One may check that the algorithm *Choose_Primary_Effects* will produce the constraint graph shown in Figure 5.4a. One of possible hierarchies generated as a linearization of the constraint graph is shown in Figure 5.4b. Based on the constraint graph, the algorithm chooses "the room is heated" as a primary effect of "use-fireplace" and "the room is lighted" as its secondary effect. This seems reasonable and corresponds to the human intuition, since we would not use the fireplace to light the room.

Suppose however that one day power went off, and electric light is no longer available. Suppose further that the day is warm, and we do not use fireplace for the purpose of heating. We may light the room using the fireplace, but as we show below, a primary-effect restricted planner would not find such a plan.

Formally, we have the initial and goal states as follows

$$S_0 = \{(\text{Light} = \text{Off}), (\text{Power} = \text{Off})\}$$
$$S_g = \{(\text{Light} = \text{On})\}$$

At level 1 of an abstraction hierarchy the goal is empty, and the only perfectly justified plan is the empty plan. While refining the plan at a concrete level, we cannot apply the operator *use-light*, because its precondition is unachievable. On the other hand, we cannot use the operator *use-fireplace*, because it changes a value of the abstract-level variable *Heat*. Thus,

operator	preconditions	effects
use-lights	(Power=On)	(Light=On)
use-fireplace	—	(Light=On), (Heat=On)

Table 5.7: The operators in the Fireplace domain

the abstract-level plan does not have an ordered refinement. □

To solve the completeness problem, we present a theorem that allows us to test whether a given hierarchy (built with primary effects) has the completeness property. First, we need to introduce the notion of a *lower-level replaceability* of an operator.

Definition 5.2 (Lower-level replaceability)
Consider an abstraction hierarchy satisfying Restriction 1a, and let S_0 be some state of the world, α be an operator whose preconditions are satisfied in S_0, and i be the criticality of the primary effects of α. The operator α is said to be lower-level replaceable *in the state S_0 if there exists a plan (S_0, S_g, Π), called a* replacing plan, *where*

$$S_g = \text{Side-Eff}(\alpha) \cup \{(x = v) \in S_0 \mid x \not\sqsubset \text{Side-Eff}(\alpha)\}$$

such that for every $\alpha_1 \in \Pi$, for every $x \sqsubset \text{Eff}(\alpha_1)$, $\text{crit}(x) < i$.

Intuitively, an operator α with i-th level primary effects is lower-level replaceable in S_0 if there exists a plan Π with the initial state S_0 that

- achieves all *side effects* of α,
- leaves all other values specified in S_0 unchanged, and
- does not establish a value of any variable whose criticality is no less than i.

For example, the operator *Move_SM* (whose side effect is *Where_S*) in the extended tower of Hanoi domain is lower-level replaceable, because we may always move the small disk without moving the other disks. On the other hand, the operator *use-fireplace* in the last example is *not* lower-level replaceable, because sometimes we cannot achieve its side effect "the room is lighted" without achieving the effect "the room is heated".

Theorem 5.3 (Completeness Condition)
A hierarchy satisfying Restrictions 1a and 2a′ has the completeness property if and only if, *for every state S and for every operator α whose preconditions are satisfied in S, α is lower-level replaceable in the state S.*

Proof. We will say that α is an i-th level operator if the criticality of the primary effects of α is i. Let us consider a hierarchy satisfying Restrictions 1a and 2a′ with n levels of abstraction, such that all operators are lower-level replaceable, and a correct concrete-level plan (S_0, S_g, Π). We need to show that there exists a sequence $\Pi_0, \Pi_1, \ldots, \Pi_{n-1}$ of plans satisfying the two properties stated in the definition of the completeness property (Definition 5.1).

In this proof we work with linear plans, and thus we start by choosing an arbitrary linearization $(S_0, S_g, \overline{\Pi})$ of (S_0, S_g, Π). Let $(S_0, S_g, \overline{\Pi}_{n-1})$ be a perfectly justified version of $(S_0, S_g, \overline{\Pi})$ on the $(n-1)$-th level of abstraction. Clearly, $\overline{\Pi}_{n-1}$ contains only $(n-1)$-th level operators. Now we use the lower-level replaceability: we replace each $(n-1)$-th level operator of $\overline{\Pi}$ that does *not* belong to $\overline{\Pi}_{n-1}$ by its lower-level equivalent. That is for each operator α_k of $\overline{\Pi}$, such that α_k is not an operator of $\overline{\Pi}_{n-1}$, we replace α_k with a corresponding replacing plan w.r.t. the state S_k, where S_k is the intermediate state of the plan $\overline{\Pi}$ before the execution of α_k. We denote the resulting concrete-level plan by $(S_0, S_g, \overline{\Pi}'_{n-1})$.

To show that the resulting plan is still correct, we observe that

- All $(n-1)$-th level preconditions and goal values are satisfied in the plan $(S_0, S_g, \overline{\Pi}'_{n-1})$ because they are satisfied in $(S_0, S_g, \overline{\Pi}_{n-1})$, and no operator of $\overline{\Pi}'_{n-1}$ that does not belong to $\overline{\Pi}_{n-1}$ changes any $(n-1)$-th level variable.

- All lower-level goal values and all lower-level preconditions of the operators remaining from the plan $\overline{\Pi}$ are satisfied because all newly inserted operators preserve the intermediate states of the old plan, by the definition of replaceability.

- Finally, the preconditions of the operators of the newly inserted replacing plans are satisfied because, by the definition of replaceability, all replacing plans are legal.

Now we consider a perfectly justified version of $(S_0, S_g, \overline{\Pi}'_{n-1})$ at the $(n-2)$-th level of abstraction. We denote this version by $(S_0, S_g, \overline{\Pi}_{n-2})$. Observe that since $\overline{\Pi}_{n-1}$ is perfectly justified on the highest level of abstraction, $\overline{\Pi}_{n-2}$ contains all operators of $\overline{\Pi}_{n-1}$. Now we use the replaceability again and replace all $(n-2)$-level operators of $\overline{\Pi}'_{n-1}$ that do *not* belong to $\overline{\Pi}_{n-2}$ by corresponding replacing plans. We denote the resulting concrete-level plan by $(S_0, S_g, \overline{\Pi}'_{n-2})$. The proof that this plan is correct is similar to the proof that $(S_0, S_g, \overline{\Pi}'_{n-1})$ is correct.

At the next step, we use the same method to build a concrete-level plan $(S_0, S_g, \overline{\Pi}'_{n-3})$, then $(S_0, S_g, \overline{\Pi}'_{n-4})$, and so on. At the i-th step, where $i \in [1..n]$, we take the concrete-level plan $(S_0, S_g, \overline{\Pi}'_{n-i+1})$ and find its perfectly justified version $(S_0, S_g, \overline{\Pi}_{n-i})$ at the $(n-i)$-th level of abstraction. Since the abstract plan $(S_0, S_g, \overline{\Pi}_{n-i+1})$, found on the previous step of the refinement process, is perfectly justified, $\overline{\Pi}_{n-i}$ contains all operators of $\overline{\Pi}_{n-i+1}$. Then we find the plan $\overline{\Pi}'_{n-i}$ by replacing all $(n-i)$-level operators of $\overline{\Pi}'_{n-i+1}$ that do not belong to $\overline{\Pi}_{n-i}$ with corresponding lower-level replacing plans. We may show that the concrete-level plan $(S_0, S_g, \overline{\Pi}'_{n-i})$ is correct the same way as we have shown that $(S_0, S_g, \overline{\Pi}'_{n-1})$ is correct.

Now observe that by construction, for every $i \in [1..n]$, the plan $\overline{\Pi}_{n-i}$ is a correct perfectly justified plan at the $(n-i)$-th level of abstraction, and all operators of $\overline{\Pi}_{n-i}$ that have effects with criticalities higher than $(n-i)$ belong to the higher-level abstract plan $\overline{\Pi}_{n-i+1}$. Thus, $\overline{\Pi}_{n-i}$ is an ordered refinement of $\overline{\Pi}_{n-i+1}$. Therefore, $\overline{\Pi}_0, \overline{\Pi}_1, \ldots, \overline{\Pi}_{n-1}$ is a sequence of plans required by the definition of the completeness property (Definition 5.1). Thus, a required sequence of plans is found.

To prove the reverse direction, we assume that, for some i, there is an i-level operator α and a state S_0 such that α is *not* lower-level replaceable in S_0. Consider the planning problem

with the initial state S_0 and the goal state

$$S_g = \textit{Side-Eff}(\alpha) \cup \{(x = v) \in S_0 \mid x \not\sqsubset \textit{Side-Eff}(\alpha) \text{ and } \textit{crit}(x) < i\}$$

In other words, we wish to achieve all side effects of α while leaving unchanged all other lower-level variables, and we do not care about the values of the variables with criticalities i and higher. This planning problem may be solved by the single-operator plan $\Pi = (\alpha)$.

Now let us consider this problem on the i-th level of abstraction. The goal on the i-th level becomes empty, and the only perfectly justified plan that solves the goal is the empty plan $\Pi_i = ()$. We claim that Π_i does not have an ordered refinement on the concrete level of abstraction.

We prove this claim by contradiction. Suppose that Π_0 is a concrete-level ordered refinement of Π_i. Then Π_0 achieves all side effects of α, and leaves all other lower-level values (that is the values of the other variables with criticalities less than i) unchanged. Since Π_0 is an ordered refinement of the empty i-level plan, Π_0 does not contain any operators that change variables with criticalities i or higher, and therefore Π_0 leaves all high-level values also unchanged. Therefore, the final state of Π_0 is

$$\begin{aligned} S_g &= \textit{Side-Eff}(\alpha) \cup \{(x = v) \in S_0 \mid x \not\sqsubset \textit{Side-Eff}(\alpha) \text{ and } \textit{crit}(x) < i\} \\ &\quad \cup \{(x = v) \in S_0 \mid \textit{crit}(x) \geq i\} \\ &= \textit{Side-Eff}(\alpha) \cup \{(x = v) \in S_0 \mid x \not\sqsubset \textit{Side-Eff}(\alpha)\} \end{aligned}$$

which by definition means that Π_0 is a replacing plan for α in the state S_0, contradicting the assumption that α is not lower-level replaceable.

Thus, the goal S_g may be achieved from the initial state S_0, but there is no corresponding perfectly justified i-th level plan that has an ordered refinement on the concrete level. Therefore, the hierarchy does *not* have the completeness property. □

Unfortunately, we did not find any method to ensure the completeness condition while generating primary-effect restricted abstraction hierarchy. Neither have we found an efficient algorithm to test whether the completeness condition holds in a given abstraction hierarchy. However, there are still two ways to use Theorem 5.3 to check the completeness of the hierarchy.

The first way is to state the stronger condition, which is sufficient but not necessary for the completeness property, but easier to check then the completeness condition. We present such a condition in the following corollary of Theorem 5.3.

Corollary 5.1 *A hierarchy satisfying Restrictions 1a and 2a′ has the completeness property if, for every operator α, there exists a correct linear plan $\overline{\Pi}$ with the initial state $S_0 = \text{Pre}(\alpha)$ and the final state*

$$S_n = \text{Side-Eff}(\alpha) \cup \{(x = v) \in S_0 \mid x \not\sqsubset \text{Side-Eff}(\alpha)\}$$

such that for any $\alpha_1 \in \overline{\Pi}$, the criticality of primary effects of α_1 is less than the criticality of primary effects of α.

Observe that S_n in the corollary is the *final state*, not only the *goal* of the plan, which means that the plan $\overline{\Pi}$ does not establish a value of any variable unspecified in S_n.

Proof. To show that a hierarchy has the completeness property, we need to prove that for any initial state S_0' and any operator α such that $\mathit{Eff}(\alpha) \subseteq S_0'$, α is lower-level replaceable in S_0'. Consider a plan $\overline{\Pi}$ with the initial and final states as specified in the statement of the corollary. We claim that the plan $(S_0', S_g', \overline{\Pi})$, where

$$S_g' = \mathit{Side\text{-}Eff}(\alpha) \cup \{(x = v) \in S_0' \mid x \not\sqsubset \mathit{Side\text{-}Eff}(\alpha)\}$$

is correct, which by definition means that α is lower-level replaceable in S_0'.

Since $S_0 \subseteq S_0'$, we conclude by Lemma 2.3 that all operators of the plan $(S_0', S_g', \overline{\Pi})$ are legal, and so it only remains to show that this plan achieves the goal. In other words, we need to show that $S_g' \subseteq S_n'$, where S_n' is the final state of $(S_0', S_g', \overline{\Pi})$. First we show that $S_n \subseteq S_g'$ by the following derivation:

$$\begin{array}{ll}
& S_n \subseteq S_g' \\
/* & \text{definitions of } S_n \text{ and } S_g' \ */ \\
\iff & (\mathit{Side\text{-}Eff}(\alpha) \cup \{(x = v) \in S_0 \mid x \not\sqsubset \mathit{Side\text{-}Eff}(\alpha)\}) \\
& \quad \subseteq (\mathit{Side\text{-}Eff}(\alpha) \cup \{(x = v) \in S_0' \mid x \not\sqsubset \mathit{Side\text{-}Eff}(\alpha)\}) \\
\iff & \{(x = v) \in S_0 \mid x \not\sqsubset \mathit{Side\text{-}Eff}(\alpha)\} \subseteq \{(x = v) \in S_0' \mid x \not\sqsubset \mathit{Side\text{-}Eff}(\alpha)\} \\
/* & \text{definition of set inclusion } */ \\
\iff & (\forall x \not\sqsubset \mathit{Side\text{-}Eff}(\alpha)) \text{ if } (x = v) \in S_0 \text{ then } (x = v) \in S_0' \\
\impliedby & (\forall x) \text{ if } (x = v) \in S_0 \text{ then } (x = v) \in S_0' \\
/* & \text{definition of set inclusion } */ \\
\iff & S_0 \subseteq S_0'
\end{array}$$

Now consider an arbitrary value $(x = v) \in S_g'$.

Case 1: $x \sqsubset S_n$. Then, since $(S_n \subseteq S_g')$, we conclude that $(x = v) \in S_n$. By Lemma 2.3, $S_n \subseteq S_n'$, and therefore $(x = v) \in S_n'$.

Case 2: $x \not\sqsubset S_n$. By Lemma 2.1, no operator of $\overline{\Pi}$ changes x, and therefore x has the same value in S_0' and S_n'. Also by Lemma 2.1, $x \not\sqsubset S_0 = \mathit{Eff}(\alpha)$, and therefore $x \not\sqsubset \mathit{Side\text{-}Eff}(\alpha)$. Now we prove that $(x = v) \in S_n'$ as follows

$$\begin{array}{ll}
& (x = v) \in S_g' \\
/* & \text{definition of } S_g' \ */ \\
\iff & (x = v) \in (\mathit{Side\text{-}Eff}(\alpha) \cup \{(x_1 = v_1) \in S_0' \mid x_1 \not\sqsubset \mathit{Side\text{-}Eff}(\alpha)\}) \\
/* & x \not\sqsubset \mathit{Side\text{-}Eff}(\alpha) \text{ and therefore } (x = v) \notin \mathit{Side\text{-}Eff}(\alpha) \ */ \\
\iff & (x = v) \in \{(x_1 = v_1) \in S_0' \mid x_1 \not\sqsubset \mathit{Side\text{-}Eff}(\alpha)\} \\
\impliedby & (x = v) \in S_0' \\
/* & x \text{ has the same value in } S_0' \text{ and } S_n' \ */ \\
\iff & (x = v) \in S_n'
\end{array}$$

Thus, we have shown that if $(x = v) \in S_g'$, then in either case $(x = v) \in S_n'$. Therefore, $S_g' \subseteq S_n'$, as desired. □

operator with abstract-level effects	replacing plan
(push-box $box $room $box-from-loc $box-to-loc)	(goto-room-loc $from $to $room)
(push-thru-dr $box $door-nm $from-room $to-room $door-loc-from $door-loc-to)	(go-thru-dr $door-nm $from-room $to-room $door-loc-from $door-loc-to)

Table 5.8: Replacing plans in the robot domain

The condition presented in the corollary may be checked by finding a plan $\overline{\Pi}$, described in the corollary, for each operator α, in other words, by solving $|O|$ planning problems. One may check that this condition holds for the extended tower of Hanoi domain. Unfortunately, the condition is too strong and often does not hold for hierarchies that satisfy the completeness property.

Another way of testing the completeness property is to use a probabilistic approach. For each operator α, we choose at random several complete initial states, and for each of chosen states check whether α is replaceable in this state. If we find that all operators are replaceable in all or almost all of chosen states, we conclude that planning in the hierarchy is near-complete.

5.5 A Robot-Domain Example

In this section we demonstrate the result of applying our algorithm to a simple *literal-represented* robot domain taken from [Yang *et al.*, 1991], which is a simplification of the domain from [Sacerdoti, 1974]. In this domain there is a robot that can walk within several rooms. Some rooms are connected by doors, which may be open or closed. In addition, there are a number of boxes, which the robot can push either within a room or from one room to another. Figure 5.5a shows an example of a robot domain. The domain may be described by the following literals:

open(d)	door d is open
box-inroom(b, r)	box b is in room r
box-at(b, loc)	box b is at location *loc*
robot-inroom(r)	the robot is in room r
robot-at(loc)	the robot is at location *loc*
location-inroom(loc, r)	location *loc* is in room r
is-door(d)	d is a door
is-box(b)	b is a box

(Observe that the last three literals are not achievable.) The list of operators in this domain, described on LISP, is given in Table 5.9.

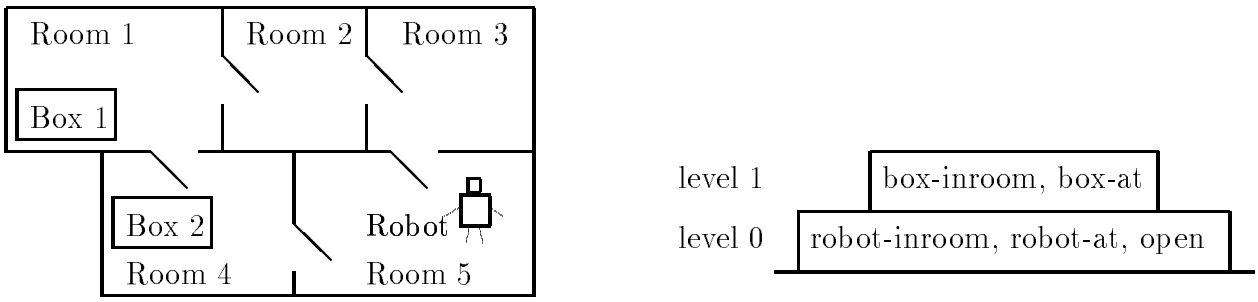
A straightforward application of Restrictions 1 and 2 to this domain fails to produce a multilevel hierarchy, while the algorithm *Choose_Primary_Effects* divides the achievable literals of the robot domain into two abstraction levels shown in Figure 5.5b. The primary effects of operators chosen by the algorithm are marked by "; **" in Table 5.9. The algorithm

```
                    ---- Operators For Moving Within a Room ----
; Go to Location within room                  ||   ; Push box between locations within a room
(setq o1 (make-operator                       ||   (setq o2 (make-operator
  :name '(goto-room-loc $from $to $room)      ||     :name '(push-box $box $room $box-from-loc
  :preconditions '(                           ||                      $box-to-loc)
     (location-inroom $to $room)              ||     :preconditions '(
     (location-inroom $from $room)            ||       (is-box $box)
     (robot-inroom $room)                     ||       (location-inroom $box-to-loc $room)
     (robot-at $from))                        ||       (location-inroom $box-from-loc $room)
  :effects '(                                 ||       (box-inroom $box $room)
     (not robot-at $from)  ;**                ||       (robot-inroom $room)
     (robot-at $to))))     ;**                ||       (robot-at $box $box-from-loc)
                                              ||     :effects '(
                                              ||       (not robot-at $box-from-loc)
                                              ||       (not box-at $box $box-from-loc)  ;**
                                              ||       (robot-at $box-to-loc)
                                              ||       (box-at $box $box-to-loc))))     ;**

                   ---- Operators For Moving Between Rooms ----
; Push box through door between two rooms     ||   ; Go through door between two rooms
(setq o3 (make-operator                       ||   (setq o4 (make-operator
  :name '(push-thru-dr $box $door-nm          ||     :name '(go-thru-dr $door-nm $from-room
                $from-room $to-room           ||                      $to-room $door-loc-from
                $door-loc-from                ||                      $door-loc-to)
                $door-loc-to)                 ||     :preconditions '(
  :preconditions '(                           ||       (is-door  $door-nm $from-room $to-room
    (is-door  $door-nm $from-room $to-room    ||                 $door-loc-from $door-loc-to)
              $door-loc-from $door-loc-to)    ||       (robot-inroom $from-room)
    (is-box $box)                             ||       (robot-at $door-loc-from)
    (box-inroom $box $from-room)              ||       (open $door-nm))
    (robot-inroom $from-room)                 ||     :effects '(
    (box-at $box $door-loc-from)              ||       (robot-at $door-loc-to)        ;**
    (robot-at $door-loc-from)                 ||       (not robot-at $door-loc-from)  ;**
    (open $door-nm))                          ||       (not robot-inroom $from-room)  ;**
  :effects '(                                 ||       (robot-inroom $to-room))))     ;**
    (not robot-inroom $from-room)             ||
    (robot-inroom $to-room)                   ||
    (not box-inroom $box $from-room)    ;**   ||
    (box-inroom $box $to-room)          ;**   ||
    (robot-at $door-loc-to)                   ||
    (box-at $box $door-loc-to)          ;**   ||
    (not robot-at $door-loc-from)             ||
    (not box-at $box $door-loc-from))) ;**    ||

                  ---- Operators For Opening and Closing Doors ----
; Open door                                   ||  ; Close door
(setq o5 (make-operator                       ||  (setq o6 (make-operator
  :name '(open $door-nm $from-room $to-room   ||    :name '(close $door-nm $from-room $to-room
              $door-loc-from $door-loc-to)    ||                $door-loc-from $door-loc-to)
  :preconditions '(                           ||    :preconditions '(
    (is-door $door-nm $from-room $to-room     ||      (is-door $door-nm $from-room $to-room
            $door-loc-from $door-loc-to)      ||                $door-loc-from $door-loc-to)
    (not open $door-nm)                       ||      (open $door-nm)
    (robot-at $door-loc-from))                ||      (robot-at $door-loc-from)
  :effects '(                                 ||    :effects '(
    (open $door-nm))))  ;**                   ||      (not open $door-nm))))   ;**
```

Table 5.9: The operators of the robot domain



(a) Example of a domain (b) Abstraction hierarchy

Figure 5.5: The robot domain

chooses the positions of boxes as abstract-level literals, and all other effects of operators as concrete-level literals.

Corollary 5.1 may be used to show that the resulting hierarchy has the monotonicity property. The replacing plans required by the conditions of the corollary, for the operators with abstract-level effects, are shown in Table 5.8. (The both replacing plans are one-operator long.) These replacing plans show that if the robot may go from one place to another with a box, it also may go without a box, and thus the secondary effects of the box-moving operators may be achieved without changing the locations of boxes.

Chapter 6

Goal-specific hierarchies

6.1 Goal-specific version of a problem domain

In this section we show that an ordered or semi-ordered abstraction hierarchy built for plans achieving a specific goal S_g may be finer-grained than a goal-independent hierarchy. The algorithm for building a goal-specific ordered abstraction hierarchy was presented in [Knoblock, 1991a]. However, Knoblock's algorithm cannot be used with our "fancy stuff" such as the learning technique and generating a primary-effected restricted hierarchy, because the algorithm presented in [Knoblock, 1991a] requires the knowledge of primary effects and forbidding preconditions of operators *before* it starts to generate a hierarchy, while our algorithms learn primary effects and forbidding preconditions in the process of the generation of a hierarchy. So, we need to adapt Knoblock's method of building a problem-specific hierarchy to our algorithms.

Definition 6.1 (Relevant Variables)
Let S_g be a goal state, and O be the set of operators in a domain. A variable x is a relevant variable *with respect to S_g if*

1. $x \sqsubset S_g$, *or*

2. *there exists an operator $\alpha \in O$ such that*
   - $x \sqsubset \mathrm{Pre}(\alpha)$
   - *there exists a relevant variable* $x_1 \sqsubset \mathrm{Eff}(\alpha)$

All operators that change relevant variables are called *relevant operators*. Observe that by definition, all precondition variables of a relevant operator are relevant. For each relevant operator α, we find its *goal-specific version* $spec(\alpha)$ by removing all its irrelevant effects. Formally, $spec(\alpha)$ is defined by

- $Pre(spec(\alpha)) = Pre(\alpha)$, and
- $Eff(spec(\alpha)) = \{(x = v) \in Eff(\alpha) \mid x \text{ is relevant}\}$

Similarly, for each state S we define its goal-specific version by removing all irrelevant variables:

$$spec(S) = \{(x = v) \in S \mid x \text{ is relevant}\}$$

The goal-specific version of a plan Π is obtained from Π by replacing all operators of Π with their goal-specific versions. Formally, there exists an *onto* function c from the set of *relevant operators* of Π onto $spec(\Pi)$ such that for any relevant operators α_1 and α_2 in Π,

(1) $c(\alpha_1) = spec(\alpha_1)$, and

(2) $c(\alpha_1) \prec_{spec(\Pi)} c(\alpha_2)$ if and only if $\alpha_1 \prec_\Pi \alpha_2$

The *goal-specific version of the domain* with respect to the goal S_g is defined by

- the set of variables $spec(\mathcal{X}) = \{x \in \mathcal{X} \mid x \text{ is relevant}\}$
- the domain of each relevant variable x, $D(x)$, the same as in the initial domain
- the goal-specific set of operators $spec(O) = \{spec(\alpha) \mid \alpha \in O \text{ and } \alpha \text{ is relevant}\}$

The main theorem of this chapter shows that, while achieving S_g, we may use the goal-specific domain instead of the initial domain without violating the correctness or completeness of planning.

Theorem 6.1 *A backward justified plan (S_0, S_g, Π) is correct if and only if its goal-specific version $(\mathrm{spec}(S_0), S_g, \mathrm{spec}(\Pi))$ is correct in the goal-specific version of the domain with respect to S_g.*

Intuitively this theorem states that, while planning, we do not need to pay attention to irrelevant variables, and we may ignore variables which are neither goal variables, nor preconditions of any relevant operator.

Proof. We divide the proof of the theorem into four claims. Throughout the proof we will write "relevant (or goal-specific)" to mean "relevant (or goal-specific) w.r.t. S_g".

Claim 1. Every operator of a backward justified plan (S_0, S_g, Π) is relevant.

Consider an arbitrary operator $\alpha \in \Pi$. Let $\overline{\Pi}$ be a linearization of Π in which α is backward justified, and $\overline{\Pi}' = (\alpha_1, \ldots, \alpha_n)$ be obtained from $\overline{\Pi}$ by the removal of all non-backward-justified operators. Then $\alpha \in \overline{\Pi}'$ and, by Theorem 3.1, $\overline{\Pi}'$ is backward justified. To show that α is a relevant operator w.r.t. the goal S_g, we prove that all operators of $\overline{\Pi}'$ are relevant. We proceed by induction from the end to the beginning of the plan, on m. The inductive hypothesis states that

$$(\forall k \in [m..n])\ \alpha_k \text{ is relevant}$$

Base: $m = n + 1$. Then the inductive hypothesis trivially holds, since the set of operators described by the hypothesis is empty.

Step: Assume that all of the operators $\alpha_{m+1}, \ldots, \alpha_n$ are relevant and consider the operator α_m. Since it is backward justified, it establishes a precondition $(x = v)$ either for the goal S_g, or for some operator α_k, where $k > m$. Since α_k is relevant, x is a relevant variable in either case by Definition 6.1, and therefore α_m is a relevant operator.

Claim 2. Let $(S_0, S_g, \overline{\Pi})$ be a (possibly incorrect) linear plan such that all operators of $\overline{\Pi}$ are relevant, and $(spec(S_0), S_g, spec(\overline{\Pi}))$ be its goal-specific version. Let $S_1, S_2, \ldots, S_n$ be the intermediate states of $(S_0, S_g, \overline{\Pi})$, and $S'_1, S'_2, \ldots, S'_n$ be the intermediate states of $(spec(S_0), S_g, spec(\overline{\Pi}))$. Then for all $k \in [1..n]$, $S'_k = spec(S_k)$.

Of course, we prove this claim by induction on k.

Base. $S_0' = \mathit{spec}(S_0)$ by the statement of the claim.

Step. Assume $S_k' = \mathit{spec}(S_k)$, and denote the k-th operator of $\mathit{spec}(\overline{\Pi})$ by ${\alpha_k}'$, that is ${\alpha_k}' = \mathit{spec}(\alpha_k)$. We derive the equality $S_{k+1}' = \mathit{spec}(S_{k+1})$ as follows:

$$
\begin{array}{l}
\quad S_{k+1}' = \mathit{spec}(S_{k+1}) \\
/* \;\; S_{k+1} = \alpha_k(S_k) \text{ and } S_{k+1}' = {\alpha_k}'(S_k') \;\; */ \\
\quad \Longleftrightarrow \;\; {\alpha_k}'(S_k') = \mathit{spec}(\alpha_k(S_k)) \\
\quad \Longleftrightarrow \;\; \mathit{Eff}({\alpha_k}') \cup \{(x = v) \in S_k' \mid x \not\sqsubset \mathit{Eff}({\alpha_k}')\} \\
\qquad\qquad = \mathit{spec}(\mathit{Eff}(\alpha_k) \cup \{(x = v) \in S_k \mid x \not\sqsubset \mathit{Eff}(\alpha_k)\}) \\
/* \;\; \text{definition of } \mathit{spec}(S) \;\; */ \\
\quad \Longleftrightarrow \;\; \mathit{Eff}({\alpha_k}') \cup \{(x = v) \in S_k' \mid x \not\sqsubset \mathit{Eff}({\alpha_k}')\} \\
\qquad\qquad = \{(x = v) \in (\mathit{Eff}(\alpha_k) \cup \{(x = v) \in S_k \mid x \not\sqsubset \mathit{Eff}(\alpha_k)\}) \mid x \text{ is relevant}\} \\
/* \;\; \text{break the left-hand side into the union of two sets} \;\; */ \\
\quad \Longleftrightarrow \;\; \mathit{Eff}({\alpha_k}') \cup \{(x = v) \in S_k' \mid x \not\sqsubset \mathit{Eff}({\alpha_k}')\} \\
\qquad\qquad = \{(x = v) \in \mathit{Eff}(\alpha_k) \mid x \text{ is relevant}\} \\
\qquad\qquad\quad \cup \{(x = v) \in S_k \mid x \not\sqsubset \mathit{Eff}(\alpha_k) \text{ and } x \text{ is relevant}\} \\
/* \;\; \text{since } \mathit{spec}(\alpha_k) = {\alpha_k}', \text{ we have } \mathit{Eff}({\alpha_k}') = \{(x = v) \in \mathit{Eff}(\alpha_k) \mid x \text{ is relevant}\} \;\; */ \\
\quad \Longleftrightarrow \;\; \{(x = v) \in \mathit{Eff}(\alpha_k) \mid x \text{ is relevant}\} \cup \{(x = v) \in S_k' \mid x \not\sqsubset \mathit{Eff}({\alpha_k}')\} \\
\qquad\qquad = \{(x = v) \in \mathit{Eff}(\alpha_k) \mid x \text{ is relevant}\} \\
\qquad\qquad\quad \cup \{(x = v) \in S_k \mid x \not\sqsubset \mathit{Eff}({\alpha_k}') \text{ and } x \text{ is relevant}\} \\
/* \;\; \text{remove the identical sets from the both sides} \;\; */ \\
\quad \Longleftarrow \;\; \{(x = v) \in S_k' \mid x \not\sqsubset \mathit{Eff}({\alpha_k}')\} = \{(x = v) \in S_k \mid x \not\sqsubset \mathit{Eff}({\alpha_k}') \text{ and } x \text{ is relevant}\} \\
\quad \Longleftrightarrow \;\; (\forall (x = v) \notin \mathit{Eff}({\alpha_k}')) \;\; (x = v) \in S_k' \\
\qquad\qquad \text{if and only if } ((x = v) \in S_k \text{ and } x \text{ is relevant}) \\
\quad \Longleftarrow \;\; (\forall (x = v)) \;\; (x = v) \in S_k' \text{ if and only if } ((x = v) \in S_k \text{ and } x \text{ is relevant}) \\
\quad \Longleftrightarrow \;\; S_k' = \{(x = v) \in S_k \mid x \text{ is relevant}\} \\
/* \;\; \text{definition of } \mathit{spec}(S_k) \;\; */ \\
\quad \Longleftrightarrow \;\; S_k' = \mathit{spec}(S_k)
\end{array}
$$

Claim 3. Let $(S_0, S_g, \overline{\Pi})$ be a correct backward justified *linear* plan. This plan is correct if and only if its goal-specific version $(\mathit{spec}(S_0), S_g, \mathit{spec}(\Pi))$ is correct.

We denote the operators of our plans as follows:

$$
\begin{array}{l}
\overline{\Pi} = (\alpha_1, \alpha_2, \ldots, \alpha_n) \\
\mathit{spec}(\overline{\Pi}) = ({\alpha_1}', {\alpha_2}', \ldots, {\alpha_n}')
\end{array}
$$

We denote intermediate states of $\overline{\Pi}$ by $S_1, \ldots, S_n$, and intermediate states of $\mathit{spec}(\overline{\Pi})$ by $S_1', \ldots, S_n'$. To prove the claim, it is enough to show that

(1) for all $k \in [1..n]$, $\mathit{Pre}(\alpha_k) \subseteq S_k$ if and only if $\mathit{Pre}({\alpha_k}') \subseteq S_k'$, and
(2) $S_g \subseteq S_n$ if and only if $S_g \subseteq S_n'$

We derive a proof of statement (1) as follows.

$Pre(\alpha_k{}') \subseteq S'_k$
/* $\alpha_k{}' = spec(\alpha_k)$, and therefore $Pre(\alpha_k{}') = Pre(\alpha_k)$ */
$\iff$ $Pre(\alpha_k) \subseteq S'_k$
/* by Claim 2, $S'_k = \{(x = v) \in S_k \mid x \text{ is relevant}\}$ */
$\iff$ $Pre(\alpha_k) \subseteq \{(x = v) \in S_k \mid x \text{ is relevant}\}$
/* by Claim 1, α_k is relevant, and therefore all its preconditions are relevant */
$\iff$ $Pre(\alpha_k) \subseteq S_k$

A proof of statement (2) is similar.

Claim 4. Let (S_0, S_g, Π) be a correct backward justified nonlinear plan. This plan is correct if and only if its goal-specific version $(spec(S_0), S_g, spec(\Pi))$ is correct.

Assume that (S_0, S_g, Π) is correct. To show that $(spec(S_0), S_g, spec(\Pi))$ is correct, we need to show that any linearization of this plan is correct. So consider an arbitrary linearization $(spec(S_0), S_g, spec(\overline{\Pi}))$. Since $(S_0, S_g, \overline{\Pi})$ is correct as a linearization of a correct plan, $(spec(S_0), S_g, spec(\overline{\Pi}))$ is correct by Claim 3. The reverse direction is proved similarly. □

6.2 Using a goal-specific domain in planning

It is straightforward to show that if criticality assignment in some domain satisfies Restrictions 1 and 2′ (or 1a and 2a), then this criticality assignment still satisfies the same restrictions in any goal-specific version of the domain. Thus, an ordered or semi-ordered hierarchy in a goal-specific version of the domain is at least as fine-grained as the corresponding hierarchy in the initial domain. On the other hand, since goal-specific versions of operators have less effects than the initial operators, a goal-specific version of the domain allows us to impose less constraints, which often leads to a finer-grained hierarchy. Thus, we may increase the number of abstraction levels by using a goal-specific domain. This method of increasing the number of levels is similar to Knoblock algorithm for generating a goal-specific hierarchy [Knoblock, 1992a].

Observe that reducing a problem domain to its goal-specific version always simplifies planning, while preserving the correctness of all plans. This means that the use of the goal-specific domain may improve the efficiency of not only abstraction techniques described in this paper, but also other planning techniques[1].

Below we describe an algorithm that finds relevant variables for a given goal S_g, and thus allows us to construct the goal-specific version of the domain. The algorithm is presented in Table 6.1. It uses the data structure called *queue* with two operations on it:

- **Add** an element x to the queue.

[1]It would be more accurate to say that the goal-specific domain never decreases the efficiency of planning: it may improve the efficiency or leave it unchanged. For example, it may be shown that the goal-specific domain does not improve the efficiency of usual non-hierarchical planning techniques that are based on backward chaining.

- **Extract** some element x from the queue (does not matter which one[2]), which means remove x from the queue and assign x to some program variable.

Both operations may be performed in constant time. For an implementation of the queue see, for example, [Cormen *et al.*, 1990].

Initially all domain variables are *white*, and upon the execution of the algorithm all relevant variables are painted *black*. For each variable x that proved to be relevant, the algorithm considers all operators that change x, and chooses all precondition variables of all these operators as relevant. x is added to the queue when x is chosen as a relevant variable and removed from the queue after the algorithm has considered all the operators changing x. Thus, every relevant variable is added to and deleted from the queue exactly once, and every irrelevant variable is never added to the queue.

Loop 1 in lines 1–2 of the algorithm paints all domain variables *white*. Loop 2 in lines 3–5 chooses all goal variables as relevant variables: the goal variables are painted *black* and added to the queue. The loop 3 in lines 6–12 extracts domain variables one by one from the queue. For each extracted variable x it considers every operator α that changes x. All preconditions of α which are still *white* are chosen as relevant: they are painted *black* and added to the queue.

Clearly, the running time of loops 1 and 2 is $O(|\mathcal{X}|)$. Loop 3 is executed once for each relevant variable x, loop 4 within it is executed once for every operator α changing x, and the running time of one execution of loop 4 is $O(|Pre(\alpha)|)$. Thus, the running time of all executions of loop 3 is

$$O(\sum_{x\in\mathcal{X}} \sum_{\alpha\in O_x} |Pre(\alpha)|) \qquad (*)$$

where O_x is the set of operators changing x. The total running time of the whole algorithm is also $(*)$. Observe that $(*)$ may be rewritten as

$$O(\sum\{|Pre(\alpha)| \ : \text{ for each pair of } \alpha \text{ and } x \text{ such that } x \sqsubset \mathit{Eff}(\alpha)\})$$

which in turn may be expressed as

$$O(\sum_{\alpha\in O} \sum_{x\sqsubset \mathit{Eff}(\alpha)} |Pre(\alpha)|)$$

and the last expression may be simplified to

$$O(\sum_{\alpha\in O} |Pre(\alpha)| \cdot |\mathit{Eff}(\alpha)|)$$

If p and e are such constants that, for any operator α, $|Pre(\alpha)| \leq p$ and $|\mathit{Eff}(\alpha)| \leq e$, then the running time of the algorithm may be expressed as $O(|O| \cdot p \cdot e)$.

---

[2]In reality, we extract the "oldest member of the queue", that is the element that was inserted before all other elements of the current queue. However, this additional information is not important for our algorithm.

```
Find_Relevant_Variables(S_g)
1.  for each x ∈ 𝒳 do  /* loop 1 */
2.      Color(x) := white;
3.  for each x ⊏ S_g do  /* loop 2 */
    begin
4.      Color(x) := black;
5.      add x to queue
    end;
6.  while the queue is not empty do  /* loop 3 */
    begin
7.      extract some x from queue;
8.      for each operator α such that x ⊏ Eff(α) do /* loop 4 */
9.          for each x_1 ⊏ Pre(α) do
10.             if Color(x)=white
                    then begin
11.                         Color(x_1) := black;
12.                         add x_1 to queue
                        end
    end
```

Table 6.1: The algorithms paints *black* all operators relevant w.r.t. S_g

6.3 Example of a goal-specific domain

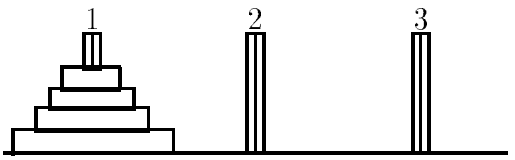
We again consider the tower of Hanoi, this time with four disks: small, medium, big, and huge (see Figure 6.1a). As usual, we may move each disk alone. Suppose that the huge disk has not only a hole in the center, but also a slit as shown in Figure 6.1c, and it can be removed from a peg or put onto a peg by pulling the disk sidewise even if there are other disks above it. This allow us to introduce an additional operator: we may move the huge disk from its peg onto any other peg a regardless of the locations of other disks. However, after doing so, we must immediately put the small disk above the same peg a[3]. Thus the additional operator allows us to move the small and huge disks from any pegs onto the common peg b. The variables in this domain are *Where_S*, *Where_M*, *Where_L*, and *Where_H*, and the formal description of the operators is presented in Table 6.2.

Suppose that we need to find a plan with the initial state as shown in Figure 6.1a, and the goal state $S_g = \{(Where_L = 2)\}$ (see Figure 6.1b). The constraint graph for this domain defined by Restrictions 1 and 2′ is as shown in Figure 6.2a. According to these constraints we have:

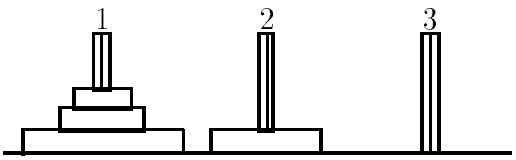
$$crit(Where_S) \leq crit(Where_M) \leq crit(Where_L) \leq crit(Where_H) \leq crit(Where_S)$$

and therefore the hierarchy based on Restrictions 1 and 2′ collapses into a single level.

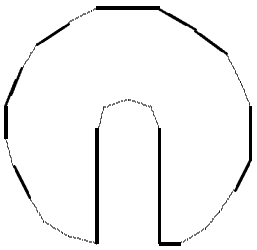
[3]A possible intuitive explanation for this restriction: assume that while pulling the huge disk sidewise, it becomes hot because of friction, and the small disk displays the warning that there may be a dangerously hot disk under it, so that nobody could accidentally burn her fingers.



(a) Initial state



(b) Goal is to put the large disk onto peg 2

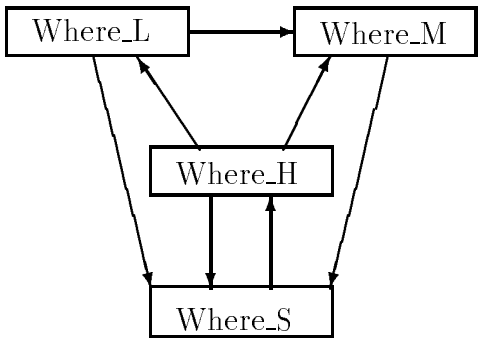


(c) Huge disk with a slit

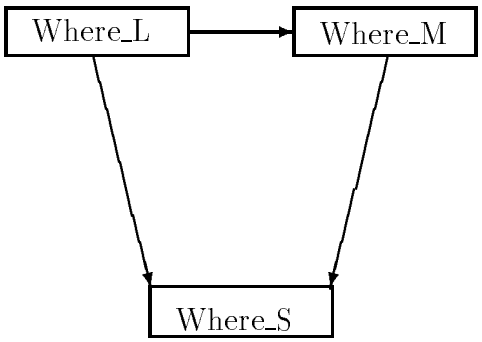
Figure 6.1: The tower of Hanoi with four disks

operator α	**preconditions** $Pre(\alpha)$	**effects** $Eff(\alpha)$
Move_S(a,b)	(*Where_S*=a)	(*Where_S*=b)
Move_M(a,b)	(*Where_M*=a), (*Where_S*≠a), (*Where_S*≠b)	(*Where_M*=b)
Move_L(a,b)	(*Where_L*=a), (*Where_S*≠a), (*Where_S*≠b) (*Where_M*≠a), (*Where_M*≠b)	(*Where_L*=b)
Move_H(a,b)	(*Where_H*=a), (*Where_S*≠a), (*Where_S*≠b) (*Where_M*≠a), (*Where_M*≠b) (*Where_L*≠a), (*Where_L*≠b)	(*Where_H*=b)
Move_SH(a)	none	(*Where_S*=a) (*Where_H*=a)

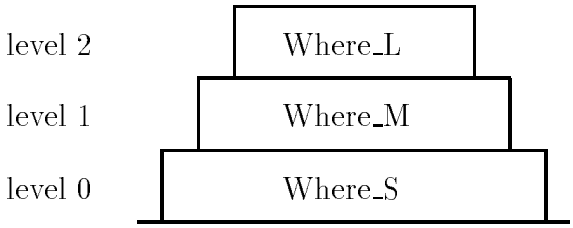
Table 6.2: The operators in the tower of Hanoi domain with four disks



(a) Constraint graph for the goal-independent domain



(a) Constraint graph for the goal-specific domain



(c) Ordered hierarchy in the goal-specific domain

Figure 6.2: Constraint graphs and the resulting hierarchy

operator α	**preconditions** $Pre(\alpha)$	**effects** $Eff(\alpha)$
$spec(Move_S(\mathrm{a},\mathrm{b}))$	$(Where_S{=}\mathrm{a})$	$(Where_S{=}\mathrm{b})$
$spec(Move_M(\mathrm{a},\mathrm{b}))$	$(Where_M{=}\mathrm{a})$, $(Where_S{\neq}\mathrm{a})$, $(Where_S{\neq}\mathrm{b})$	$(Where_M{=}\mathrm{b})$
$spec(Move_L(\mathrm{a},\mathrm{b}))$	$(Where_L{=}\mathrm{a})$, $(Where_S{\neq}\mathrm{a})$, $(Where_S{\neq}\mathrm{b})$ $(Where_M{\neq}\mathrm{a})$, $(Where_M{\neq}\mathrm{b})$	$(Where_L{=}\mathrm{b})$
$spec(Move_SH(\mathrm{a}))$	none	$(Where_S{=}\mathrm{a})$

Table 6.3: The goal-specific version of the tower of Hanoi domain with four disks

One of the possible ways to increase the number of effects is to generate a primary-effect restricted hierarchy. Another way, which we are going to use in this example, is to build the goal-specific domain w.r.t. S_g.

The variable *Where_L* is the only goal variable, and at the first step, the algorithm *Find_Relevant_Variables* chooses it as a relevant variable. The only operator that changes *Where_L* is *Move_L*, and at the next step the algorithm chooses the still-not-chosen precondition variables of *Move_L*, which are *Where_S* and *Where_M*, as relevant variables. Then the algorithm considers the operators changing *Where_S*, which are *Move_S* and *Move_SH*, and the operator *Move_M*, which changes *Where_M*, and finds that the preconditions of these operators do not contribute any new relevant variables. The operator *Where_H* does not establish any of relevant variables, and so the algorithm does not consider it. Thus, the only relevant variables are *Where_S*, *Where_M*, and *Where_L*. Intuitively this means that while moving the small, medium, or large disk, we do not need to worry about the position of the huge disk.

The corresponding problem-specific versions of operators are shown in Table 6.3. The constraint graph for these operators is shown in Figure 6.2b, and the corresponding hierarchy in Figure 6.2c. The hierarchy contains three levels, which is better than a single-level hierarchy in the initial domain.

Chapter 7

Conclusion

7.1 Summary

This thesis extends previous work, most notably Knoblock's, on a theory of abstractions. The main results of the thesis are the following.

- The thesis formalizes the notion of plan justification, describes the different kinds of justified plans and the algorithms for finding justified versions of a given plan, and shows that the task to find the optimal justification of a given plan is NP-complete.

- We have shown the connection between the notions of plan justification and ordered abstraction hierarchies, and demonstrated that different definitions of justification lead to different kinds of ordered hierarchies. Semi-ordered hierarchies, introduced in the thesis, preserve all advantages of ordered hierarchies, but may have more levels of abstraction. We presented algorithms for generating these finer-grained hierarchies in polynomial time.

- The technique of forbidding-restricted and primary-effect restricted planning allows us to increase the efficiency of many planners (not only hierarchical planners) by avoiding deadends in the search space and reducing the branching factor.

- Syntactic conditions of the ordered property of primary-effect restricted hierarchies capture the intuition behind "good" hierarchies in primary-effect restricted planning and enable us to find automatically primary effects of operators. Primary-effect restricted hierarchies are finer-grained than hierarchies generated by ALPINE and often allow us to build a multi-level hierarchies in the cases when ALPINE's hierarchy collapses into a single level. The thesis introduces the completeness property and presents necessary and sufficient conditions of this property for primary-effect restricted hierarchies. The technique of automatically finding primary effects and insuring completeness of primary-effect restricted planning may also be used in non-hierarchical planning.

- The notion of goal-specific domains generalizes the technique of building problem-specific ordered hierarchies [Knoblock, 1991a] and enable us to improve the efficiency

of planning by tailoring *any* abstraction technique (not only ordered hierarchies) and, generally, any planning strategy to a specific goal state.

7.2 Future work

Both theoretical and experimental work is required to evaluate the efficiency of the techniques suggested in this thesis. We have conducted a few experiments in simple domains to check usefulness of described algorithms, but more experiments are required to demonstrate the general efficiency of the discussed techniques. A theoretical work similar to [Bacchus and Yang, 1992] may be done to evaluate the efficiency of planning in semi-ordered and primary-effect restricted hierarchies and give mathematical description of domains in which these hierarchies may increase the efficiency of planning. It may be interesting to perform controlled experiments to compare our algorithms and ALPINE with algorithms that use different methods for generating abstraction hierarchies, such as ABSTRIPS [Sacerdoti, 1974], PABLO [Christensen, 1990], and an algorithm based on the downward refinement property [Bacchus and Yang, 1991]. This direction of research is closely related to a much more general open problem of developing a theory for evaluating expected running time of different planning algorithms.

Another question that may be addressed from both experimental and theoretical points of view is comparative "optimality" of different kinds of plan justifications. For example, while we have shown that backward justification is weaker than well-justification, we still do not know how often the justification of a given plan found by the *Backward_Justification* algorithm is shorter than the justification found by *Well_Justification*, and whether the difference is significant.

More work is required on applications of the notion of domain rules. Three important algorithms that may be implemented are

- to check whether a given set of domain rules is an invariant,
- to find the set of domain rules for a given set of initial-state axioms, and
- to find all (or some) invariants in a problem domain

A possible application of domain rules is to reduce the search during the planning process by detecting states that do not satisfy domain rules and avoiding these states. Also, domain rules may be used to increase the number of levels while generating abstraction hierarchies [Knoblock, 1991a]. The domain rules are used by some planners, such as PRODIGY [Carbonell *et al.*, 1991] and ALPINE [Knoblock, 1991a], and it would be useful to generate domain rules automatically.

We mentioned in Subsection 4.2.2 how to use the notion of forbidding operators in implementing a forbidding-restricted planner to avoid deadends during the search for a correct plan. This allows us to reduce the search space without violating the completeness of planning. This method may be used with many planning techniques (not only hierarchical) to increase their efficiency. A possible direction of future work is to implement a forbidding-restricted planner.

The purpose of our algorithm for selecting primary effects is to maximize the number of strongly connected components, which means that we take the number of abstraction levels as the only measure of the "goodness" of a primary-effect restricted hierarchy. However,

other measures of goodness may be introduced. One intuitive observation in this direction is that an even distribution of domain variables among abstraction levels may be important. For example, if a problem domain has n variables, it is probably better to have two levels of abstraction, each containing $n/2$ variables, then three levels of abstractions containing respectively 1, 1, and $(n-2)$ variables. (Some theoretical support for this intuition may be derived by the method presented in [Knoblock, 1991b]). Also, it would be interesting to approach the problem of generating hierarchies with the Downward Refinement Property [Bacchus and Yang, 1991] using primary-effect restricted planning.

If the same primary-effect restricted hierarchy is used repeatedly, it may be worthwhile to find a hierarchy with *the maximal* possible number of levels using the A^* technique [Nilsson, 1980] instead of our greedy algorithm. Also it may be possible to use A^* for finding a perfectly justified subplan of a given plan. The third, more interesting and probably much more complicated problem concerning A^* is to adapt an A^*-based planner for finding the optimal plan using a hierarchy of abstraction spaces. Also, it may be possible to design efficient algorithms for finding a perfect justification of a plan and for building a hierarchy with the maximal number of levels in some special types of problem domains, such as unary and postunique domains.

The methods of finding primary effects suggested in the thesis are purely syntactic. An open problem is to find the semantical meaning of primary effects, and describe ways of choosing primary effects that formalize human intuition behind the "main result" of an action.

While the use of a primary-effect restricted planner is a powerful technique for increasing the efficiency of planning, we have to ensure the completeness property of a hierarchy in order to use this technique without violating completeness of planning. This poses the problem of designing efficient methods for checking whether a given primary-effect restricted hierarchy has the completeness property and, if not, restoring completeness by imposing additional constraints. This may be done by an application of a learning technique similar to the technique presented in Subsection 4.3.1. On the other hand, it may be possible to ensure the completeness property during the generation of a hierarchy.

Further research in the directions suggested in this section may lead to developing new methods of abstraction in planning and unifying the existing abstraction techniques with a general theory of hierarchical planning.

Appendix A

Summary of Notation

Below we summarize the notation used in the thesis. For each letter and abbreviation used as an element of notation, the second column indicates the page where the notation is introduced, and the third column briefly explains what this element stands for.

notation	page	meaning
x	4	variable
$(x = v)$	4	the variable x has the value v
$(v \neq x)$	5	the variable x has a value different from v
$D(x)$	4	domain of x (the set of the values that x can accept)
$D_a(x)$	7	set of all achievable values of x
$D_u(x)$	8	set of all unachievable values of x
$\mathcal{X}$	4	set of all variables in a planning domain
$\mathcal{X}_c$	8	set of all changeable variables in a planning domain
$\mathcal{X}_u$	8	set of all unchangeable variables in a planning domain
S	4	state of the world
$\|S\|$	5	the number of values specified in the state S
$S(x)$	4	value of the variable x in the state S
S_0	8	initial state
S_g	8	goal state
S_m	8	intermediate state of a linear plan after the first m operators
S_n	8	final state of a linear plan
α	5	operator
$Pre(\alpha)$	5	preconditions of the operator α
$\|Pre(\alpha)\|$	6	the number of preconditions of α

notation	page	meaning
$\mathit{Eff}(\alpha)$	5	effects of the operator α
$\lvert\mathit{Eff}(\alpha)\rvert$	6	the number of effects of α
$\mathit{Prim\text{-}Eff}(\alpha)$	70	primary effects of the operator α
$\mathit{Side\text{-}Eff}(\alpha)$	70	side effect of the operator α
$\mathit{Out}(\alpha)$	7	outcomes of the operator α
$\alpha(S)$	7	state achieved by applying the operator α to the state S
O	7	set of all operators in a planning domain
$O_{(x=v)}$	7	set of all operates that achieve the value v of x
O_x	7	set of all operators that change the variable x
$\overline{\Pi}$	8	linear plan
$\overline{\Pi}(S)$	8	state achieved by applying the linear plan $\overline{\Pi}$ to the state S
Π	8	nonlinear plan
$\alpha_1 \prec_\Pi \alpha_2$	8	time-precedence relation: α_1 must be executed before α_2 in Π
$\lvert\Pi\rvert$	9	the number of operators in the plan Π
P	9	the number of preconditions of all operators in a plan plus the number of values in its goal S_g: $P = \sum_{\alpha\in\Pi} \lvert\mathit{Pre}(\alpha)\rvert + \lvert S_g\rvert$
E	9	the number of effects of all operators in a plan: $E = \sum_{\alpha\in\Pi} \lvert\mathit{Eff}(\alpha)\rvert$
l	13	literal
$\neg l$	13	negation of the literal l
$\mathcal{L}$	13	set of all literals in a literal-represented planning domain
r	17	domain rule
I	17	invariant
A	17	initial-state axioms
R	17	domain rules
H	20	abstraction hierarchy
$\mathit{crit}(x)$	20	criticality of the variable x
${}_iS$	21	set of the values with criticalities i and more in the state S
${}_i\mathit{Pre}(\alpha)$	21	preconditions of α at the i-th level of abstraction
${}_i\alpha$	21	i-th level version of α, with preconditions ${}_i\mathit{Pre}(\alpha)$ and effects $\mathit{Eff}(\alpha)$
${}_iO$	21	set of the i-th level versions of all operators in a problem domain
$\mathit{spec}(S)$	90	goal-specific version of the state S
$\mathit{spec}(\alpha)$	89	goal-specific version of the operator α

Appendix B

List of algorithms

Below we present a brief descriptions of all discussed algorithms in the order as they occur in the article. For each algorithm we indicate its *average-case* running time.

Check_Preconditions_1(S, α), page 23
Check whether the preconditions of the operator α are satisfied in the state S, for the full representation of a problem domain. The running time is $O(|Pre(\alpha)|)$.

Apply_1(S, α), page 23
Find the state resulting from the application of the operator α to the state S, for the full representation of a problem domain. The running time is $O(|Eff(\alpha)|)$.

Check_Preconditions_2, page 24
Check whether the preconditions of the operator α are satisfied in the state S, for the closed-world representation of a problem domain. The running time is $O(|Pre(\alpha)| \cdot \log |S|)$.

Apply_2, page 24
Find the state resulting from the application of the operator α to the state S, for the closed-world representation of a problem domain. The running time is $O(|Eff(\alpha)| \cdot \log |S|)$.

Minimal_Elements(G), page 28
Find the minimal elements of the partially ordered set represented by the directed graph G. The running time is $O(|G|)$, where $|G|$ is the number of vertices in the graph, which is the same as the number of elements in the corresponding partially ordered set.

Backward_Justification(S_0, S_g, Π), page 33
Find a backward justified version of the correct plan (S_0, S_g, Π). The running time is $O(E \cdot |\Pi|^2)$.

Possibly_Establish$(\alpha,\alpha_1,(x = v))$, page 33
Check whether the operator α possibly establishes the precondition $(x = v)$ for the operator α_1 in some plan Π. The running time is $O(|\Pi|)$.

Well_Justification, page 38
Find a backward justified version of the correct plan (S_0, S_g, Π). The running time is $O(P \cdot |\Pi|^4)$.

Legal_Plan(S_0, Π), page 38
Check whether the plan Π with the initial state S_0 is a legal plan. The running time is $O(P \cdot |\Pi|^2)$.

Legal_Operator(S_0, Π, α), page 38
Check whether the operator α is legal in the plan Π with the initial state S_0. The running time is $O(P \cdot |\Pi|)$.

Greedy_Justify_Checking(S_0, S_g, Π, α), page 43
Check whether the operator α in the plan (S_0, S_g, Π) is greedily justified. The running time is $O(P \cdot |\Pi|^3)$.

Illegal_Operators(S_0, S_g, Π), page 43
Find all illegal operators of the plan (S_0, S_g, Π). The running time is $O(P \cdot |\Pi|^2)$.

Greedy_Justification, page 44
Find a greedily justified version of the plan (S_0, S_g, Π). The running time is $O(P \cdot |\Pi|^5)$.

Linear_Greedy_Justification$(S_0, S_g, \overline{\Pi})$, page 45
Find a greedily justified version of the correct linear plan (S_0, S_g, Π). The running time is $O((P + E) \cdot |\Pi|^2)$ for the full representation of states and $O(P \cdot |\Pi|^2 \cdot \log |S_n|)$ for the closed-world representation.

Remove_Abstract_Effects(Π', Π, i), page 50
Remove all abstract-effect operators from the plan Π, where Π is an i-th level refinement of the higher-level plan Π'. The running time is $O(E)$.

Forbidding_Test(O, α), page 58
Perform the Second Forbidding Test to find (some of) forbidding preconditions of α. The running time is $O(|Pre(\alpha)|^2 \cdot \sum_{\alpha_1 \in O_{(x=v)}} (|Eff(\alpha)| + |Pre(\alpha)|))$.

Impose_Restriction_1, page 64
Create the constraint graph for the variables in a problem domain and add edges corresponding to Restriction 1 to the graph. The running time is $O(\sum_{\alpha \in O} |Eff(\alpha)| + |\mathcal{X}|^2)$.

Check_Restriction_2''(S_0, S_g, Π), page 65
Find operators of the plan (S_0, S_g, Π) that do not satisfy Restriction 2''. If such operators have been found, impose additional constraints to make Restriction 2'' hold. The running time is $O(E \cdot |\Pi|)$ plus the time necessary to add new constraints.

Add_Constraint(x_1, x), page 65
Add an edge from x_1 to x to the constraint graph and modify the abstraction hierarchy accordingly. The running time is $O(|\mathcal{X}|^2 \cdot \log \log |\mathcal{X}|)$.

Choose_Primary_Effects, page 73

Choose primary effects of the operators in a planning domain and build a primary-effect restricted hierarchy. The running time is $O(|\mathcal{X}|^2 \cdot \sum_{\alpha \in O} |\mathit{Eff}(\alpha)| + |O| \cdot |\mathcal{X}|^2 \cdot \log\log|\mathcal{X}|)$.

User-defined_Primary_Effects, page 74
Impose constraints for the primary effects of operators determined by a user. The running time is $O(\sum_{\alpha \in O}(|\mathit{Pre}(\alpha)| + |\mathit{Eff}(\alpha)|) + |\mathcal{X}|^2)$.

Choose_Primary_Effect_Of_Operator(α), page 75
Choose primary effects of α so as to maximize the number of strongly connected components in the constraint graph. The running time is $O(|\mathit{Eff}(\alpha)| \cdot |\mathcal{X}|^2 + |\mathcal{X}|^2 \cdot \log\log|\mathcal{X}|)$.

Make_Value_Be_Primary_Effect($x = v$), page 76
Make $(x = v)$ primary effect of some operator, choosing the operator so as to maximize the number of strongly connected components in the constraint graph. The running time is $\sum_{\alpha \in O_{(x=v)}}(|\mathit{Eff}(\alpha)| \cdot |\mathcal{X}|^2 + |\mathcal{X}|^2 \cdot \log\log|\mathcal{X}|)$.

Choose_Primary_Effects_According_To_Graph, page 77
Create the explicit set of the primary effects of each operator according to the restrictions defined by the constraint graph. The running time is $O(\sum_{\alpha \in O} |\mathit{Eff}(\alpha)|)$.

Find_Relevant_Variables(S_g), page 94
Find the set of relevant variables in a problem domain w.r.t. the goal S_g. The running time is $O(\sum_{\alpha \in O} |\mathit{Pre}(\alpha)| \cdot |\mathit{Eff}(\alpha)|)$.

Bibliography

[Aho *et al.*, 1972] Alfred V. Aho, M. R. Garey, Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput*, 1(2), pages 131–137, 1972.

[Aho *et al.*, 1974] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[Bacchus and Yang, 1991] Fahiem Bacchus and Qiang Yang. The downward refinement property. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, pages 286–291, 1991.

[Bacchus and Yang, 1992] Fahiem Bacchus and Qiang Yang. The expected value of hierarchical problem-solving. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

[Backstrom and Klein, 1991] Christer Backstrom and Inger Klein. Parallel non-binary planning in polynomial time. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, pages 286–291, 1991.

[Carbonell *et al.*, 1991] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. PRODIGY: an integrated architecture for planning and learning. Research Report, School of Computer Science, Carnegie Mellon University, 1991. Tech. Report CMU-CS-CS-89-189.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32, pages 333-377, 1987.

[Christensen, 1990] Jens Christensen. A hierarchical planner that generates its own abstraction hierarchies. In *Proceedings of Eighth National Conference on Artificial Intelligence*, pages 1004–1009, 1990.

[Cormen *et al.*, 1990] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to algorithms.* MIT Press, 1990.

[Enderton, 1977] Herbert B. Enderton. *Elements of set theory.* Academic Press, California, 1977.

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, pages 189–208, 1971.

[Fink and Yang, 1992a] Eugene Fink and Qiang Yang. Formalizing plan justifications. *Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI)*, pages 9–14, 1992.

[Fink and Yang, 1992b] Eugene Fink and Qiang Yang. Automatically abstracting the effects of operators. *Proceedings of the First International Conference on AI Planning Systems*, pages 243–251, 1992.

[Knoblock, 1990] Craig A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 923–928, 1990.

[Knoblock, 1991a] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving.* Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1991. Tech. Report CMU-CS-91-120.

[Knoblock, 1991b] Craig A. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 686–691, 1991.

[Knoblock *et al.*, 1991] Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 692–697, 1991.

[Korf, 1985] Richard E. Korf. *Learning to solve problems by search for macrooperators.* Pitman Publishing Inc, Marshfield, Massachusetts, 1985.

[Minton *et al.*, 1991] Steven Minton, John Bresina, and Mark Drummond. Commitment strategies in planning: a comparative analysis. *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, pages 259–261, 1991.

[Newell and H. Simon, 1972] Allen Newell and Herbert A. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Nilsson, 1980] Nils J. Nilsson. *Principles of artificial intelligence.* Tioga Pub. Co., Palo Alto, CA, 1980.

[Sacerdoti, 1974] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2), pages 115–135, 1974.

[Sacerdoti, 1975] Earl D. Sacerdoti. The nonlinear nature of plans. *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, page 206–214, 1975.

[K. Simon, 1985] Klaus Simon. *An improved algorithm for transitive closure on acyclic digraphs.* Technical Report A85/13, Universitat des Saarlandes, Fachbereich 10, 1985.

[Tate, 1977] Austin Tate. Generating project networks. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, 888-893, 1977.

[Tenenberg, 1988] Josh D. Tenenberg. *Abstraction in Planning.* Ph.d. Thesis, University of Rochester, Dept of Computer Science, 1988. Tech. Report 250.

[Wilkins, 1984] David E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22(3), pages 269–301, 1984.

[Yang and Tenenberg, 1990] Qiang Yang and Josh Tenenberg. ABTWEAK: abstracting a nonlinear, least commitment planner. In *Proceedings of Eighth National Conference on Artificial Intelligence*, pages 923–928, Boston, MA, 1990.

[Yang *et al.*, 1991] Qiang Yang, Josh Tenenberg, and Steven Woods. *Abstraction in nonlinear planning.* University of Waterloo, Waterloo, August 1991. Research Report CS-91-65.

Index

abstract-effect operator, 53
abstraction
    hierarchy, *see* hierarchy
    level, 23
    space, 22
ABSTRIPS planner, 2
achievable value, 9
achieving a value, 8
acyclic graph, 26
adjacency list, 27
ALPINE planner, 2, 50
applying an operator, 8, 16

backward justification, 33
backward justified
    operator, 33
    plan, 33, 34
backward ordered
    hierarchy, 52
    refinement, 52
backward semi-ordered
    hierarchy, 54
      refinement, 53
black-red tree, 26

changeable variable, 9
changing a variable, 8
Closed-World Assumption, 26
closed-world representation, 26
combining connected components, 29
completeness condition, 90
completeness property, 88
complete state, 5, 15
constraint graph, 68
correct
    plan, 9, 10
    subplan, 11
criticality, 22

directed edge, 26
directed graph, 26
domain
    goal-specific version of, 98
    of a variable, 5
    postunique, 18
    unary, 18
    water-boiling, 7
domain rules, 18
    preserving, 21

edge, 26
    transitive, 27
effects, 6, 15
    primary, 2, 73, 75
    side, 2, 73, 75
establishing a value, 8
establishment, 33
    possible, 33

finer-grained hierarchy, 2, 23
first
    forbidding test, 61
    non-forbidding test, 60
forbidding preconditions, 55
forbidding-restricted planner, 56
forbidding test, 60
    first, 61
    second, 61
full representation, 25

goal-specific version
    of a domain, 98
    of an operator, 97
    of a state, 98
    of a plan, 98
goal state, 1, 9
GPS planner, 2

graph, 26
  acyclic, 26
  constraint, 68
  directed, 26
  transitively closed, 27
  transitively reduced, 27
  vertex of, 26
greedily justified
  operator, 45
  plan, 47
greedily ordered hierarchy, 64
greedily semi-ordered
  hierarchy, 65
  refinement, 64
greedy justification, 45

hierarchical problem solving, 50
hierarchy, 1, 22, 23
  finer-grained, 2, 23
  ordered, 50, 52, 64
    backward, 52
    greedily, 64
    perfectly, 64
    well-, 64
  primary-effect restricted, 76
  semi-ordered, 54, 65
    backward, 54
    greedily, 65
    perfectly, 65
    well-, 65

immediate predecessors, 29
immediate successors, 29
initial state, 9
initial-state axioms, 20
invariant, 20

justification, 32
  backward, 33
  greedy, 45
  perfect, 40
  well- 33, 37
justified operator, *see* operator
justified plan, *see* plan

legal operator, 8, 16
legal plan, 9, 10
level of abstraction, 23
linearization, 10, 29
linear plan, 9
literal-representation, 15
literal, 2, 15
lower-level replaceability, 90

minimal element, 30
monotonicity property, 88

necessary precedence, 10
negating a value, 8
non-forbidding test, 60
  first, 60
  second, 60
nonlinear plan, 10

operator, 1, 6
  abstract-effect, 53
  applying, 8, 16
  goal-specific version of, 97
  justified
    backward, 33
    greedily, 45
    well-, 37
  legal, 8, 16
  relevant, 97
  type, 14
ordered
  hierarchy, 50, 52, 64
    backward, 52
    greedily, 64
    perfectly, 64
    well-, 64
  refinement, 52
outcomes, 8

partial state, 6, 15
path in a graph, 26
perfect justification, 40
perfectly justified plan, 40
perfectly ordered hierarchy, 64
perfectly semi-ordered
  hierarchy, 65
  refinement, 64

plan, 1, 9
  correct, 9, 10
  goal-specific version of, 98
  justified, 2
    backward, 33, 34
    greedily, 37
    perfectly, 40
    well-, 33, 37
  legal, 9, 10
  linear, 9
  nonlinear, 10
  replacing, 90
planner
  forbidding-restricted, 56
  linear, 6
  primary-effect restricted, 2, 73, 75
possible establishment, 33
possible precedence, 10
postunique domain, 18
precedence, 10
  necessary, 10
  possible, 10
preconditions, 6, 15
  forbidding, 55
preserving a rule, 21
primary-effect restricted hierarchy, 76
primary-effect restricted planner, 2, 73, 75
primary effects, 2, 73, 75
proper subplan, 11

queue, 101

refinement, 51
  ordered, 52
    backward, 52
  semi-ordered, 53, 64
    backward, 53
    greedily, 64
    perfectly, 64
    well-, 64
relevant
  operator, 97
  variable, 97
replaceability, 90
replacing plan, 90
representation
  closed-world, 26
  full, 25
Restrictions
  Restriction 1, 52
  Restriction 2, 52
  Restriction 2′, 56
  Restriction 2″, 67
  Restriction 1a, 76
  Restriction 2a, 76
  Restriction 2a′, 78
rule, *see* domain rule

second
  forbidding test, 61
  non-forbidding test, 60
semi-ordered
  hierarchy, 54, 65
    backward, 54
    greedily, 65
    perfectly, 65
    well-, 65
  refinement, 53, 64
    backward, 53
    greedily, 64
    perfectly, 64
    well-, 64
side effects, 2, 73, 75
state, 5, 6
  complete, 5, 15
  goal, 1, 9
  goal-specific version of, 98
  initial, 9
  partial, 6, 15
strongly connected components, 29
  combining, 29
subplan, 10
  correct, 11
  proper, 11

test
  forbidding, 60
    first, 61
    second, 61
  non-forbidding, 60

first, 60
second, 60
3-CNF, 42
3-CNF-SAT problem, 42
time-precedence relation, 10
topological sorting, 29
tower of Hanoi, 13
transitive closure, 30
transitive edge, 27
transitively closed graph, 27
transitively reduced graph, 27
transitive reduction, 30

unary domain, 18
Upward Solution Property, 23

value
achievable, 9
achieving, 8
establishing, 8
negating, 8
variable
changeable, 9
changing, 8
domain of, 5
relevant, 97
variable-representation, 15
vertex of a graph, 26

water-boiling domain, 7
well-justification, 33, 37
well-justified
operator, 33
plan 33, 37
well-ordered hierarchy, 64
well-semi-ordered
hierarchy, 65
refinement, 64