

Directions In Planning: Understanding The Flow Of Time In Planning

Bart Massey

CIS-TR-99-??
June 1999

Abstract

Since the notion of general purpose planning became one of the touchstones of artificial intelligence, surprisingly little improvement has been made in the efficiency of planning algorithms. Efficient high-speed search is essential to most planning algorithms proposed to date: this can only be achieved if the search algorithms used are based on a solid understanding of the search space. The concept of search directionality—searching temporally or causally forward or backward—has been quite important to designers of planning algorithms. Nonetheless, this concept appears to be poorly understood.

Through a series of constructions and experiments, it is shown here that successful planners must be capable of both forward chaining and backward chaining behavior, and that understanding directionality issues in planning is a necessary precursor to the construction of efficient planners.

This work begins by discussing some of the underpinnings of directionality in planning: the physical, psychological, and computational temporal arrows that directionally orient planning problems. A previously unappreciated property of directionality in planning is then described, namely that the direction of planning problems is not a fundamental property of the standard formalism. Planning problems can be reversed, allowing a planner to search in the opposite of its normal direction without a change in planning algorithm. Next, a novel technique is described for determining the directional behavior of existing planners, and experimental results using an implementation of this technique are reported. This analysis of planner direction can be used to better understand the search strategy used in modern planning algorithms such as satisfiability-based planning. Finally, some consequences and extensions of the results are given.

Together, these results shed new light on the construction of planning algorithms using high-speed search, and thus move us closer to making planning practical for real problems.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Copyright 1999 Bart Massey

This Technical Report is a reformatted version of the author's Doctoral Dissertation, completed June 1999. Examining committee:

Dr. Matthew Ginsberg, Co-chair
Dr. Amr Sabry, Co-chair
Dr. Christopher Wilson
Dr. John Orbell

Dedication

To my father, Dr. William H. Massey. His dedication to science and to his family inspired my life's work, and his tragic death has left a void in my life that will never be filled.

Acknowledgements

Thank God, from whom all blessings flow. Thanks to my wife, Joanie, for putting up with me, and to my baby Benjamin, just for being.

Thanks to my advisor, Matt Ginsberg, who set me on this track and kept me moving forward. Matt encouraged me to share my small insights of chapter 2, challenged me to derive and prove the results of chapter 3, and conceived the need for an extrinsic planner directionality test that eventually resulted in the work of chapters 4 and 5. Finally, when I was slacking, he bullied me into the serious work of writing it all down. This dissertation would have been impossible to complete without him.

Thanks to: Amr Sabry, for co-chairing my committee. Cynthia Brown, without whose support over the past six months this project could not have been completed. The current and past members of the University of Oregon Computational Intelligence Research Laboratory (CIRL), including Matt, Jimi Crawford, Brian Drabble, David Etherington, David Joslin, Dave Clements, Tania Bedrax-Weiss, and Andrew Parkes, for many great ideas and useful discussions. David Etherington, for his detailed reviews of chapter 3. Dave Clements, for reading early drafts. The faculty, staff, and students of the University of Oregon Computer and Information Sciences Department. The CIRL funders, whose support was vital. Finally, all the other folks, too numerous to list, who provided help, advice, and encouragement.

This work was sponsored in part by a grant from the Air Force Office of Scientific Research (agreement number F49620-93-I-0572), and by grants from the Defense Advanced Research Projects Agency and the Air Force Research Laboratory, Rome, NY (agreement numbers F30602-95-1-0023, F30602-97-1-0294, and F30602-98-2-0181). The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFOSR, DARPA, Rome Labs, or the U.S. Government.

Contents

1	Introduction	1
1.1	History and Background	2
1.1.1	Planners	2
1.1.2	Early Planners	3
1.1.3	Means-End Analysis	3
1.1.4	Persistence	3
1.2	STRIPS Planning	4
1.2.1	Propositional STRIPS Planning	4
1.2.2	Complexity of PROPS	7
1.2.3	Protection Conditions and Causal Links	8
1.2.4	Predicate STRIPS Planning	8
1.2.5	Extensions to STRIPS	9
1.3	Comments	11
2	The Metaphysics Of Directionality	12
2.1	Directionality in Planning	13
2.1.1	Time’s Arrow: Directionality and Physics	13
2.1.2	The Mind’s Eye: Directionality and Thought	13
2.1.3	The Electronic Brain: Directionality and Algorithmics	15
2.2	Confounding Issues	16
2.2.1	Directionality and Hierarchy	16
2.2.2	Directionality and Abstraction	17
2.2.3	The Limits of STRIPS	17
2.3	Conclusion	18
3	STRIPS Problem Reversal	19
3.1	PROPS Actions and the Frame Axiom	19
3.1.1	Reversibility and Deleted Preconditions	20
3.2	Reversal Using Compilation	20
3.2.1	C_2 : Positive-Only Preconditions	21
3.2.2	C_3 : “Don’t Care” Effects	25
3.2.3	C_R : Reversal	26
3.2.4	Partial Goal States and Reversal	30
3.3	Extensions and Reversibility	30

3.4	Conclusions	31
4	Determining The Directionality Of Planners	33
4.1	Search Space Planning	33
4.1.1	Search Direction	34
4.1.2	Propagation	36
4.1.3	Propagating Planners	38
4.2	One-Way Functions	39
4.3	Boolean Circuits	39
4.3.1	NBC Planning Problems	40
4.3.2	NBCPPs and Reversal	42
4.4	A Planner Directionality Test	43
4.5	Construction	43
4.5.1	Bit Commitment	44
4.5.2	Bit Commitment Network	45
4.5.3	Final Construction	47
4.6	Correctness	49
4.7	Conclusions	50
5	Measuring Planner Direction	52
5.1	Implementation	52
5.2	Experiments	54
5.2.1	Methodology	54
5.2.2	Directionality of ASP	56
5.2.3	Directionality of O-Plan	58
5.2.4	Directionality of UCPOP	60
5.2.5	Directionality of Graphplan	60
5.2.6	Directionality of Blackbox	64
5.3	Parallel Plans	67
5.3.1	Graphplan	70
5.3.2	Blackbox Using WSAT	70
5.3.3	Blackbox Using Relsat	72
5.4	Overall Results	75
6	Directions In Planning	77
6.1	“Don’t Know” and “Don’t Care” in PROPS Actions	77
6.1.1	Unrestricted DC Effects	77
6.1.2	DC Effects and the Initial State	78
6.1.3	Explicit DC Action Effects	79
6.1.4	“Don’t Know” Effects	81
6.1.5	DC and DK Conditions	82
6.2	Composing One-Way Planning Problems	83
6.3	Tractability of Circuit-Based Planning	86
6.4	Summary: Expressiveness and Directionality of STRIPS	86

6.5	Review	87
6.6	Future Work	88
6.7	Conclusions	89

List of Tables

3.1	Possible a_0, a_1 (\hat{a}_0, \hat{a}_1) Pairs for C_2	24
3.2	Possible Pairs for C_3	27
3.3	Possible a_0, a_1 Pairs for C_R	29
3.4	Possible \hat{a}_0, \hat{a}_1 Pairs for C_R	29
4.1	Truth Table for Left Projection Gate	46
6.1	Possible a_0, a_1 Pairs for $C_{?*r}$	84
6.2	Possible \hat{a}_0, \hat{a}_1 Pairs for $C_{?*r}$	84

List of Figures

3.1	Scheme C_2 : Compilation to Positive-Only Preconditions	22
3.2	Scheme C_3 : Extending C_2 to 3-Valued Logic	26
3.3	Scheme C_R : Reversal Rules	27
4.1	One-Way Function Planning Problems	44
4.2	One-Way Planning Problem with Bit Commitment	45
4.3	Operators for Left Projection Gate	46
4.4	Schematic for Left Projection Gate	46
4.5	Gate and Fluent Labeling by Bit/Layer	47
4.6	4-Bit Commitment Network	48
4.7	Forward One-Way Planning Problem	48
4.8	Backward One-Way Planning Problem	49
5.1	One-Way Function \mathcal{H}_n	53
5.2	Detector Testbed Organization	55
5.3	ASP Directional Performance: Time	57
5.4	ASP Directional Performance: Search Time	57
5.5	ASP Directional Performance: Nodes	58
5.6	O-Plan Directional Performance: Time	59
5.7	O-Plan Directional Performance: Nodes	59
5.8	UCPOP Directional Performance: Time	61
5.9	UCPOP Directional Performance: Nodes	61
5.10	Graphplan Directional Performance: Graph Nodes	62
5.11	Graphplan Directional Performance: Time	63
5.12	Graphplan Directional Performance: Search Nodes	63
5.13	Blackbox/Graphplan Directional Performance: Time	65
5.14	Blackbox/WSAT Directional Performance: Time	66
5.15	Blackbox/Relsat Directional Performance: Time	66
5.16	Blackbox/Relsat Directional Performance: Relsat Time	67
5.17	Blackbox/Relsat Directional Performance: Variables Valued	68
5.18	Parallel Graphplan Directional Performance: Graph Nodes	70
5.19	Parallel Graphplan Directional Performance: Time	71
5.20	Parallel Graphplan Directional Performance: Search Nodes	71
5.21	Parallel Blackbox/WSAT Directional Performance: Time	72
5.22	Parallel Blackbox/WSAT Directional Performance: WSAT Time	73

5.23	Parallel Blackbox/WSAT Directional Performance: WSAT Flips	73
5.24	Parallel Blackbox/Relsat Directional Performance: Time	74
5.25	Parallel Blackbox/Relsat Directional Performance: Relsat Time	74
5.26	Parallel Blackbox/Relsat Directional Performance: Variables Valued	75
6.1	Reversal Rules for Restricted DC Effects	80
6.2	Scheme C_4 : Extending C_3 to 4-Valued Logic	81
6.3	Scheme $C_{?*r}$: Full Reversal Rules	83
6.4	Bidirectional Planning Problem	85
6.5	Island Planning Problem	85

Chapter 1

Introduction

One of the traditional distinctions in planning is that between *forward chaining* from temporally or causally earlier states to later states and *backward chaining* from later to earlier states. Modern planning algorithms often exhibit both behaviors, but almost all are biased in one direction or the other. The earliest planners [38] tended to search forward in state space, while the next generation of planners [41] tended to use backward means-end analysis.

It is, however, surprisingly difficult to pin down notions of forward and backward chaining. It would be useful to understand the real differences in planning algorithms that are motivated by this distinction. Since planning is known to be intractable in general [14], it is difficult to distinguish efficiency of planning algorithms by worst-case complexity arguments. It is also hard to produce indisputably “real-world problems” under the simplified assumptions of Propositional STRIPS Planning (PROPS).

The fundamental claims presented in this work are that

- Successful planners must at least be capable of both forward chaining and backward chaining behavior.
- Understanding directionality issues in planning is a necessary precursor to the construction of efficient planners.

Several results are presented in support of these claims.

- Various external and internal considerations relevant to directional chaining in planning are surveyed, in order to understand the scope and nature of the problem.
- It is shown how to construct the reversal of a STRIPS problem, such that forward chaining on the reversed problem corresponds to backward chaining on the original problem (and *vice versa*).
- It is shown how to construct a class of STRIPS problems that have the property that they cannot be tractably solved by one-directional branching. These results are used experimentally to explore the directionality of several existing planning algorithms, by analyzing performance differences of planner implementations on forward and backward tractable problems without reference to the underlying planning algorithm.

This work proceeds as follows: The remainder of chapter 1 gives a brief background in planning, concentrating on PROPS. Chapter 2 discusses some of the philosophy and history of directionality in STRIPS. Chapter 3 describes an algorithm for transforming a STRIPS problem into an isomorphic problem such that planning in one direction in the transformed instance corresponds to planning in the other direction in the original instance. Chapter 4 describes how to construct STRIPS problems that are tractable in only one direction, and chapter 5 reports on empirical results in using these problems to determine the directionality of existing STRIPS planners. Chapter 6 clears up some loose ends, suggests some topics for future research, and summarizes some general conclusions one can draw from this work.

1.1 History and Background

The ability to formulate a complex multi-step plan for achieving a goal is one of the characteristics that separates humans from even the highest animals. Chimpanzees have been observed to form plans of a few steps [13]. However, compared to their skills in planning, their skill in language and tool use, two other oft-cited measures of intelligence, seem quite profound.

Given this association of intelligence with planning skill, it was natural for the pioneers of artificial intelligence (AI) to turn to planning as an AI problem domain. In addition, the practical advantages of computerized planning were obvious early on; some of the first work in planning concentrated on “automatic programming” [56], in which a computer constructs its own program given a specification of the problem to be solved.

1.1.1 Planners

The notion of planning is difficult to pin down in its details, but most definitions of planning would agree on the general outline of the problem: Given a current state of the world (*initial state*), a desired state of the world (*goal state*), and a collection of methods that change the world state (*actions*), a *generative planner* is a mechanism that selects actions transforming the initial state to the goal state. Actions are described in terms of the world states in which they can be executed (*preconditions*), and the world states they can produce when executed (*effects*).

It is useful to distinguish between action descriptions or *types* as part of a planning problem description, and action *instances* that are performed during execution. In this work, action types will often be referred to as *operators* and action instances as actions. Thus, a single operator in a planning problem description may correspond to many actions in a plan produced by a planner. However, the common convention will be adopted of referring to operators as actions in situations where the context is unambiguous.

Kambhampati [27] describes the useful distinction between classical planning and more arcane planning methods in terms of restrictions on the world states and actions of the planning problem. The classical planning restrictions can be summarized as follows:

- All world states are discrete, complete, unchanging, and freely and fully observable.
- All actions are atomic, deterministically conjunctive in their preconditions, and deterministic in their effects.

In particular, classical planning problems exclude those forms of *hierarchical* planning problem in which some sort of action or domain hierarchy is given as part of the problem description. It is generally believed that hierarchical problem decomposition is an important part of the human planning process. In classical planning, the hope is that hierarchical information can be extracted by the planning algorithm from the classical description. In the meantime, there is plenty of challenge to be had in non-hierarchical domains.

This work is concerned only with classical planning, and largely with restricted versions of classical planning. Nonetheless, the problem is considered hard, and it is widely hoped that progress will lead to insight into the solution of more general planning problems.

1.1.2 Early Planners

The collections of papers edited by Allen, Hendler, and Tate [1] provides an introduction to the early history of planning. Historically, research has moved from powerful and general conceptions of planning and planners toward progressively simpler notions (although recently there has been much research into planning in probabilistic domains).

The earliest work on planning treated it as a form of automated theorem proving: the available action types were given as axioms, and the theorem prover was asked to build a constructive proof that the goal state was reachable from the initial state. Because the descriptions of states and actions were so general, largely as a result of being expressed in such powerful formalisms, this sort of planning was intractable for any realistically sized problem. However, attempts to understand the computational difficulties led to notions about the planning process that are still in widespread use, such as means-end analysis.

1.1.3 Means-End Analysis

One important consideration arising from the study of human reasoning in planning has been the concept of *means-end analysis*. The fundamental idea behind this concept is that human reasoning contains a strong notion of cause and effect. It is reasonable to suppose that humans plan by choosing a particular goal to be achieved (the *end*) and then selecting an action that achieves it (the *means*).

This notion has been adopted by nearly all AI planning systems as a method of controlling plan generation. A typical planner attempts to choose goals (elements of the goal description) and subgoals (intermediate goals arising during the planning process) in a reasonable order, satisfying each by means of an action or action sequence.

Some researchers (e.g., McDermott [36]) have questioned whether means-end analysis is the best approach for planner control. Nonetheless, the large amount of experience gained with this mechanism and its intuitive appeal suggest that it will continue to be a dominant choice in planning algorithms.

1.1.4 Persistence

One of the principal sources of inefficiency in theorem-proving planners arises from what McCarthy and Hayes called the *frame problem* [35]. The problem, which in this work will be termed the *persistence problem*, is this: In a purely logical formulation of planning one must explicitly enforce

the condition that elements of the world state will not change spontaneously (i.e., except as the result of action execution). Unfortunately, enforcing persistence can require quite general axioms in a first-order formulation, or many axioms in a propositional formulation. In either case, reasoning about plans may become intractable due to this complexity.

1.2 STRIPS Planning

The advent of Fikes and Nilsson’s 1971 Stanford Research Institute Problem Solver (STRIPS) [17] marks a major change in the view of planning problems. In the STRIPS formalism, the frame axioms that encode persistence are made implicit in the formalism instead of explicit.

STRIPS introduces the notion of *add lists* and *delete lists*: an action changes the world state by setting the values of state variables true or false. Once set, these values persist until changed by some other action. This approach to the persistence problem allows a planner to avoid having to prove theorems about state variables unaffected by an action: instead, the planner can automatically infer that the values of these variables are the same before and after the action.

McDermott [36] provides a good general introduction to the state of STRIPS planning as of 1994. Short of imposing some restrictions on the formalism in which the preconditions and effects of actions can be expressed, STRIPS planning today is not terribly different from STRIPS planning in 1971.

1.2.1 Propositional STRIPS Planning

Perhaps the simplest notion of planning currently of interest to AI researchers is Propositional STRIPS Planning (PROPS). In PROPS, world states are given by the values of a finite set of Boolean state variables (*fluents*). The preconditions of an operator are given as a list of atomic formulae over the fluents, each of which must hold in order for the action to be executed. The effects of the operator are given as a list of fluents that will be forced true by the execution of the action (the *add list*), and a list of fluents that will be forced false by the execution of the action (the *delete list*).

The remainder of this work will adopt a standardized notation similar to that of Bylander [14] for PROPS planning problems. The principal difference between the formalism of this work and that of Bylander is that this work introduces the notion of *legal* actions and plans. Bylander follows a notion originally due to McCarthy [35] in which an action may be executed in any world state, but has no effect unless its preconditions are satisfied. This is convenient in some ways, but it will be important in what follows in Chapters 3 and 4 to understand the conditions under which a plan is legal.

A PROPS planner works in a domain consisting of operators whose preconditions and effects are expressed in terms of propositional conjunctions of ground state variables (*fluents*). The preconditions and effects of operators, as well as the initial and goal states of a problem, will be represented by *conjunctive formulae* over the fluents:

Definition 1.1 (Element-wise Negation) *The element-wise negation of a set S is denoted \bar{S} and defined by*

$$\bar{S} = \{\neg e \mid e \in S\} \cup \{e \mid \neg e \in S\}$$

This syntactic definition of negation leads naturally to the notion of a set as a consistent collection of atoms.

Definition 1.2 (Consistent Set) A set S is consistent if and only if $\nexists e \in S . \neg e \in S$.

Thus, a consistent set of atoms can be thought of as a conjunction.

Definition 1.3 (Conjunctive Formulae) The set of conjunctive formulae over a set S is denoted $cf(S)$ and defined by

$$cf(S) = \{s \subseteq S \cup \bar{S} \mid s \text{ is consistent} \}$$

The set of conjunctive formulae with error over S is denoted $cf_{\perp}(S)$ and defined by

$$cf_{\perp}(S) = cf(S) \cup \{\perp\}$$

with the distinguished element \perp denoting an error condition.

The above definitions provide a basis for the formal definition of operators.

Definition 1.4 (PROPS Operator) A PROPS action type or operator is a tuple $a = \langle \text{pre}, \text{eff} \rangle$, where pre is a conjunctive formula denoting the preconditions of a , and eff is a conjunctive formula denoting the effects of a . (Thus the effects of a can be the empty formula. Such no-op operators are usually not included in real domain descriptions.) An action is the instantiation of an operator in a plan.

It is conventional to specify the fluents added to or deleted from the world state by an action.

Definition 1.5 (PROPS Additions/Deletions) The additions $\text{add}(a)$ of an action a collect the fluents that occur positively in $\text{eff}(a)$. The deletions $\text{del}(a)$ collect the fluents whose negations occur in $\text{eff}(a)$. Specifically, $\text{add}(a) = \text{eff}(a)$, and $\text{del}(a) = \overline{\text{eff}(a)}$.

For example, an action with $\text{eff}(a) = \{f, \neg g\}$ would have additions $\text{add}(a) = \{f, \neg g\}$ and deletions $\text{del}(a) = \{\neg f, g\}$.

The definition of action execution then closely follows Bylander's, but with an explicit error state:

Definition 1.6 (PROPS Action Execution) The state $a(s)$ resulting from executing action a in state s is defined by

$$a(s) = \begin{cases} (s \setminus \text{del}(a)) \cup \text{add}(a) & \text{pre}(a) \subseteq s \\ \perp & \text{otherwise} \end{cases}$$

An action a is legal in a state s if and only if $\text{pre}(a) \subseteq s$.¹

It is not a part of the definition of action execution that the result of executing a legal action in a consistent state is another consistent state. One would desire and expect such a property to hold, and indeed it does.

¹Note that since \perp is not a set, $\text{pre}(a) \subseteq \perp$ can never be true: thus, once the error state has been produced, all further action execution will preserve the error state. This is a bit confusing, but the alternatives are notationally cumbersome.

Proposition 1.1 *Let a be a legal action in a consistent state s . Then $a(s)$ is consistent.*

PROOF: *By contradiction. If $a(s)$ is inconsistent, there must be some fluent f such that $f \in a(s)$ and $\neg f \in a(s)$. There are three cases.*

s does not mention f : By definition 1.6 the only way to achieve $f \in a(s)$ and $\neg f \in a(s)$ in this case is if $f \in \text{eff}(a)$ and $\neg f \in \text{eff}(a)$. But since by definition 1.4 $\text{eff}(a) \in \text{cf}(F)$, this is impossible, hence a contradiction.

$f \in s$: Since s is consistent, it must be that $\neg f \notin s$. Thus, by definition 1.6 $\neg f \in \text{eff}(a)$. But by definition 1.5 this implies that $f \in \text{del}(a)$, so by definition 1.6 $f \notin a(s)$. Hence a contradiction.

$\neg f \in s$: Since s is consistent, it must be that $f \notin s$. Thus, by definition 1.6 $f \in \text{eff}(a)$. But by definition 1.5 this implies that $\neg f \in \text{del}(a)$, so by definition 1.6 $\neg f \notin a(s)$. Hence a contradiction.

In this work, operators will be written using a horizontal bar, with the conjunctive formula specifying the preconditions above the bar, and the conjunctive formula specifying the effects below. A PROPS operator that moves a block a from a block b to a block c in a standard formulation of the infamous “blocks-world” domain might look like

$$\frac{\text{clear-}a \text{ clear-}c \text{ on-}a\text{-}b}{\text{clear-}b \text{ on-}a\text{-}c \text{ } \neg\text{on-}a\text{-}b \text{ } \neg\text{clear-}c}$$

This action can only be executed in a state in which block a is clear, block c is clear, and block a is on block b . After the action is executed, block b will be clear, block a will be on block c , block a will be clear and not on block b , and block c will no longer be clear.² An operator *mentions* any fluent that appears in the precondition or the effect of the operator, regardless of sign; thus this operator mentions five different fluents.

Given the above, it is straightforward to define PROPS planning domains, planning problems, and plans.

Definition 1.7 (PROPS Domain, Problem) *A PROPS domain is a tuple $\langle F, A \rangle$, where F is a set of fluent symbols and A is a set of operators. A problem is a tuple $\langle D, I, G \rangle$ where D is a domain and I and G are conjunctive formulae over the fluents F of D representing initial and goal states respectively. A state S (commonly, the goal state G) is partial if there is a fluent in the domain mentioned neither positively nor negatively in S . In this work, states are total unless otherwise specified.*

Definition 1.8 (PROPS Plan) *A plan is any sequence of actions a_1, \dots, a_n . A plan is executed in a state s by executing each of its actions in turn*

$$(a_1, \dots, a_n)(s) = \begin{cases} (a_2, \dots, a_n)(a_1(s)) & n > 0 \\ s & \text{otherwise} \end{cases}$$

A plan ρ is legal in a state s if and only if $\rho(s) \neq \perp$. A plan is valid in a state s if and only if it is legal and satisfies the goal (i.e., $G \subseteq \rho(s)$).

²That the blocks a , b , and c are all different blocks is implicit in the fact that this is a ground rule: if a , b , and c were variables, explicit uniqueness conditions would be needed. See pp. 8–9 below for a discussion of Predicate STRIPS.

A plan is legal if and only if all of its actions are legal.

Proposition 1.2 *A plan*

$$\rho = a_1, \dots, a_n$$

is legal in a state s if and only if $s \neq \perp$ and

$$\forall i \in \{1, \dots, n\} . a_i \text{ is legal in } (a_1, \dots, a_{i-1})(s)$$

PROOF: *By induction on the length n of ρ .*

Base case:

For $n = 0$ the quantified condition is trivially true, and since $\rho(s) = s$, ρ is legal if and only if $s \neq \perp$.

Inductive case:

Assume that the proposition is true for $n = j$. $\rho(a_{j+1}) \neq \perp$ if and only if action a_{j+1} is legal in state $s_j = (a_1, \dots, a_j)(s)$. If a_1, \dots, a_j is illegal in s , then $s_j = \perp$ and thus a_{j+1} is illegal in s_j by the definition of action execution. But if a_1, \dots, a_j is legal in s , then $s_j \neq \perp$ and by the definition of action execution a_{j+1} will be legal in s_j if and only if $\text{pre}(a_{j+1}) \subseteq s_j$. Thus the proposition holds for $n = j + 1$ as well.

1.2.2 Complexity of PROPS

The classic analysis of PROPS complexity is the work of Bylander [14]. He shows that even under a variety of strong restrictions to the already restrictive PROPS model planning is PSPACE-complete in measures of the size of the problem description, such as number of operators or number of fluents.

The intuition behind this result is that the length of a plan in actions can be exponential in the number of fluents in the state. The best known example of this is the Towers of Hanoi problem [24] with n disks. This can be encoded as a PROPS planning problem with $\mathcal{O}(n^2)$ fluents, indicating for each pair of disks whether the first is on top of the second, and $\mathcal{O}(n^3)$ operators, one for each possible move of the first disk from the second to the third. The minimum-length solution to the problem requires $\mathcal{O}(2^n)$ actions.

To prove that PROPS planning is PSPACE hard, it is sufficient to observe that any DTM transition can be encoded as a planning operator (for details, see Bylander [14]). Checking the correctness of a PROPS plan can be done with just one bit per fluent; starting with the initial state, one simply checks that the preconditions of each action hold, and then modifies the state according to the effects of the action. If (and only if) the last action produces a goal state, then the plan solves the problem. Thus the PROPS planning problem must be in PSPACE, hence it is PSPACE complete.

There are restrictions to PROPS planning that may increase its tractability. Foremost, it is evident that fixing a polynomial length limit on plans (i.e., asking the related question “does there exist a plan of length $\mathcal{O}(n^k)$ that solves this PROPS planning problem?”) brings the complexity of planning into NP. Indeed, PROPS planning can be proven to be NP-complete under this restriction. Bylander [14] gives a few restrictions on the form of operators that will make the problem

NP-complete, although this is surprisingly difficult. Ginsberg [22] shows that PROPS planning becomes NP-complete if the requirement that all preconditions of operators be met is relaxed slightly. In general, however, planning appears to be of high complexity—a discouraging result for AI researchers.

1.2.3 Protection Conditions and Causal Links

In the classical planning domain, a fluent’s value may be thought of as caused by the effects of a particular action. Indeed, a common tactic in formalizing classical planning is to regard the initial conditions as effects of some dummy action that must begin the plan, and the goal as the preconditions of some dummy action that must end the plan. This notion is a natural expression of the notion of means-end reasoning (p. 3).

Another natural outgrowth of means-end reasoning is the *protection condition* or *causal link*; an annotation on a plan indicating that action A’s effect causes a particular value of a fluent f needed by action B’s preconditions. Since causality flows forward in time, A must precede B. Since B is expecting A to produce its precondition, during the interval between them the value of f must not be changed. If the annotation is regarded as a protection condition, actions whose effect on f are the same as A’s will naturally be allowed—a phenomenon sometimes referred to as *shadowing*. If the annotation is regarded as a causal link, shadowing will naturally be disallowed. Both choices are reasonable, with complex tradeoffs between them.

1.2.4 Predicate STRIPS Planning

Propositional STRIPS is a useful tool for algorithmic analysis and design of planning systems. In practice, it has proven to be insufficiently expressive to easily encode realistic planning domains. Instead, a formulation known as Predicate STRIPS (PREDS) has become the standard for STRIPS-style planning.

Except for the persistence assumption, the original STRIPS formulation strongly resembled the axiomatic first-order logical planning methods that preceded it. Later work indicated that the formalism could be much simplified without significantly reducing its expressive power.

This simplified PREDS formalism, much easier to understand and control, looks much like PROPS, except that effects and preconditions are not restricted to conjunctions of ground atomic formulae. Instead, they are conjunctions of predicate formulae, and are implicitly universally quantified. The details differ from system to system in generally uninteresting ways: It is sufficient to consider “flat predicates” with variables quantified over a set of ground objects. By allowing variables to be shared between the preconditions and effects of an action, it becomes possible to succinctly describe a class of actions using these *action schemata*.

Thus, a blocks-world planning problem might have just three action schemata:

$$\begin{array}{c}
 \frac{\neg table(Y) \ \neg table(Z) \ on(X, Y) \ clear(X) \ clear(Z)}{\neg on(X, Y) \ on(X, Z) \ clear(Y) \ \neg clear(Z)} \\
 \frac{table(Y) \ \neg table(Z) \ on(X, Y) \ clear(X) \ clear(Z)}{\neg on(X, Y) \ on(X, Z) \ \neg clear(Z)} \\
 \frac{\neg table(Y) \ table(Z) \ on(X, Y) \ clear(X)}{\neg on(X, Y) \ on(X, Z) \ clear(Y)}
 \end{array}$$

Note that the initial state must be extended as well; in addition to a specification of the actual blocks world problem, the type predicate *table* must be defined as well. Without a closed world assumption, this will require a large number of ground initial conditions:

$$\begin{array}{l}
 table(TABLE) \\
 \neg table(a) \\
 \neg table(b) \\
 \dots
 \end{array}$$

This is awkward, and a simple extension to PREDs that removes this requirement is discussed under the heading of “Types” below.

1.2.5 Extensions to STRIPS

For convenience, as well as for increased expressive power in some cases, a number of extensions to the STRIPS formalism have been proposed and implemented. Many of them are included as options to the PDDL [37] problem description language. Several of the most common extensions are described here, and will be considered in subsequent chapters.

Closed World Assumption: It is often convenient to assume that any fluent that is not explicitly mentioned in a state description is false: this is the so-called “Closed-World Assumption (CWA)” that is implicit in much of the planning literature.

For PROPS, this assumption is useful primarily for specifying the initial state—in practice the set of initially true fluents is often much smaller than the set of all fluents. (The goal state typically is specified by a partial state description: fluents not specified may be either true or false in the goal state.) Indeed, simple fluent renaming permits recoding of an arbitrary PROPS problem description so that all fluents are false in the initial state. For some systems (such as Graphplan [8]) neither the initial state nor the goal description may have explicitly false fluents. This will in general require extra fluents to translate an arbitrary problem description (as described on pp. 21–25 of chapter 3), but does not change the expressiveness or complexity of PROPS planning [14].

For PREDs, the CWA is more problematic. It is usually stated in such a way that all variables range over finite sets of ground values: this is the “Closed World” referred to. In this situation, the CWA of the previous paragraph still holds.

Types: Under the CWA, the standard PREDS formulation has the odd property that the closed world is implicit, rather than being explicitly described. Further, no distinction is made between arguments of the various predicates. For example, there is no distinction between the table and any other block in the blocks world description above, except that provided by the awkward introduction of a “timeless” predicate whose sole purpose is to define the table. A more natural formulation of PREDS allows the explicit declaration of typed objects in the domain, and thus permits the use of implicit type predicates.

Conditional Effects: In realistic environments, the effect of an action differs depending on the situation in which it is executed. Ginsberg and Smith [23] were among the first to comment on this, giving various examples from a household robot environment. They point out that objects which should sometimes move together present particular problems for STRIPS planners, since an exponential number of operators are required to correctly handle all the possible subsets of objects that may need to move as a unit. While proper indexing allows the correct operator to be selected in linear time (or better), the large size of the problem description is generally viewed as making this approach intractable.

The standard solution to this problem is the *conditional effect* [39]: allow specified additions and deletions in an action to be conditional upon properties of the preceding state. Conditional effects are straightforward for most planners to deal with, and easy for planning domain engineers to understand, making this a popular addition to STRIPS.

Domain Axioms: PREDS implementations normally require some explicit representation of world states: in PROPS, fluent values are usually stored explicitly. This can be problematic for two reasons. First, the size of the representation can become large, and thus difficult to reason with. Second, action effects can become quite complicated in attempting to maintain this complex state information.

In the blocks world, for example, the *clear* fluents are normally maintained for each block by each block *move* action. It is desirable, however, for this maintenance to be performed by invoking the obvious rule: a block is clear if no block is on top of it. A *domain axiom* can enforce exactly this condition: the *clear* fluents become derivatives of the world state, rather than a part of it.

It can be a bit tricky to give a sound and useful semantics for domain axioms that is nonetheless tractable for most planners. Indeed, domain axioms written in full first-order logic may not be tractable.

Safety Constraints: A *safety constraint* is a mechanism for ensuring that a specified condition holds in all states of a plan (this is the *dont-disturb* constraint of Weld and Etzioni [58]). As the name implies, this mechanism is suitable mainly for ensuring that a plan never includes a state with undesired consequences. Support for safety constraints appears to be easy to add to existing planners, but the utility of the mechanism is still somewhat unproven.

This is only a small selection of the many extensions that have been proposed to STRIPS planning: as noted above, these extensions are considered mainly in understanding how the research results presented in this work might apply to modern general-purpose planners.

1.3 Comments

As noted above, AI researchers have been working on planning for about 30 years, and on classical planning for about 20 years. In this time, they have made surprisingly little progress. With the possible exception of Wilkins' SIPE [59], the author is aware of no domain-independent classical planning engine in use in a fielded application. In limited toy domains, such as blocks world, the latest general-purpose classical planners are able to provide provably optimal solutions to problems that humans probably could not [29]. However, humans can provide solutions to these problems reasonably quickly when permitted to neglect optimality considerations, unlike existing automatic mechanisms.

Work relating the properties of planning algorithms, problem domains, and the STRIPS formalism is essential to identifying and expanding classes of tractable classical planning problems. The recent success of Graphplan [9] in achieving dramatic speedups on a large class of benchmark problems energized the entire AI planning community. Unfortunately, it is still poorly understood how Graphplan achieves this level of success—it appears that this planner serves as sort of a Rorschach test for planning researchers, who see in it exactly what they desire and expect to see.

The remainder of this work takes a different approach to improving planning performance. Selecting a particular feature of STRIPS planning, namely the distinction between forward and backward chaining planners, this distinction is explored in terms of the interaction between the formalism, planners, and problems. Some important results are established, showing ultimately that unidirectional planners will have poor performance on certain classes of tractable problems, and giving a method for transforming such problems to increase their tractability for simple planners.

One can hope that a similar approach, applied to other STRIPS issues, will eventually lead to high-performance planners built from first principles, driven by science rather than engineering. Only time can tell whether this hope will come to fruition.

Chapter 2

The Metaphysics Of Directionality

From the early history of AI planning, there has been an important distinction made between forward and backward chaining planners [20]. This distinction is based on the observation that, at least superficially, the forward flow of time in plan execution appears to induce an asymmetry that distinguishes the direction of chaining. The distinction is further supported by human introspection into planning, and by empirical results showing that forward and backward chaining planners, even when otherwise superficially similar, often have dramatically different performance characteristics [5].

As a result of these intuitions about directionality, AI researchers have made some fairly strong assumptions about the advantages of backward chaining. Perhaps typical is Kambhampati's comment [27] that

Compared to forward state-space refinement [planning], the backward state-space refinement generates plan sets with a fewer number [*sic*] of components because it concentrates only on those actions that are relevant to current goals. This focus on relevant actions, in turn, leads to a lower branching factor for planners that consider the plan set components in different branches.

Kambhampati apparently has the intuition that the number of subgoals active at a given time will be small in real-world problems, and thus backward-chaining will produce a narrower search tree. One can construct artificial problems in which the forward search tree is narrow and the backward search tree large: apparently Kambhampati believes that these are not realistic. This sort of comment is hard to evaluate: it is difficult to separate the facts from the hypotheses.

McDermott [36] suggests that

... working backward and forward are not done symmetrically. Planning in the forward direction starts with a complete situation description, and after computing the result of every action, retains a complete description. Planning in the backward direction proceeds from a goal statement that merely constrains one aspect of the final situation. This requires solving the *regression problem*.

The suggestion is that the asymmetry between forward and backward planning is the result of the asymmetry between the total initial state and the partial goal description. The truth is somewhat deeper than that: the natural formulation of regression leads to partial intermediate states even in the presence of a total goal state.

It seems clear that the role of time direction in planning is poorly understood. The remainder of this chapter attempts to build a framework that will make it (somewhat) easier to understand the role of time in planning and execution. This should facilitate the presentation in the chapters that follow. The putative advantage of backward chaining will then be discussed in the light of this framework.

2.1 Directionality in Planning

There are three major reasons why planning might appear to be temporally asymmetric: the fact of temporal asymmetry in the physical world, temporal asymmetry in the thought processes of humans, and the appearance of an apparent temporal asymmetry in the STRIPS formalism itself. In this section, each of these factors will be considered in turn, to determine their role in the overall picture of STRIPS temporal asymmetry.

2.1.1 Time's Arrow: Directionality and Physics

Physicists have a reasonable understanding of the role of the flow of time in the physical universe. The flow of time is often characterized in terms of statistical laws known as the Laws of Thermodynamics, which state that certain changes in physical systems normally proceed forward in time. In particular the Second Law of Thermodynamics [57, pp. 562–568] states that the entropy of a physical system, a measure of its disorder, will tend to increase over time. Observable consequences of the law of entropy include such things as the tendency of objects to reach equilibrium temperature with their surroundings and the inclination of liquids to run downhill.

Thus, the forward temporal direction may be defined as the direction in which entropy tends to increase. From a planning perspective, the key question is whether the STRIPS formalism is expressive enough to capture physical notions of entropy, and thus to capture this temporal arrow. If STRIPS captured physical entropy, it would mean that planning for an entropically illegal action (e.g., running water uphill) would be impossible in the formalism. Clearly, at least as the formalism is normally used to encode the physical world, this is not the case: one can write a “run-water-uphill” operator just as easily as a “run-water-downhill” one.

The science of information theory [42] also has a notion of entropy. While it is often confused with the thermodynamic notion, it is at its core almost entirely different. The principal tenuous connection between the two notions of entropy is a theoretical lower bound on the amount of the energy transferred in transmitting a single bit of information. For the purposes of this work, the key distinction here is that information-theoretic entropy appears to carry no temporal arrow: there is no obvious notion of information flowing in a particular time direction. Thus, it would be a mistake to characterize planning as directional because of informational entropy.

2.1.2 The Mind's Eye: Directionality and Thought

The only known working example of an intelligent system is the biological brain. How do human and animal brains handle time direction in planning? The evidence is far from clear. The most popular view (held by, e.g., Ginsberg¹) is that human planners work backward from the goals to be

¹Personal communication, 1998.

achieved, selecting from the range of operators available to produce each subgoal. Proponents of this view point out what means-end analysis is supposed to capture: in real-world problems there are usually a large number of actions available in any given world state, and yet there are often few actions that will achieve a specific subgoal. It is unarguable that some form of goal directed search must be employed by humans to avoid searching through an unmanageably large number of possible plans: humans cannot effectively manage many plans simultaneously.

On the other hand, humans appear to be better at reasoning about the state of the world after an action (given the state before) than at solving the regression problem. Faced with a puzzle-mode problem, the tendency appears to try to solve it from the beginning to the end. For example, trying to solve the Peg Solitaire puzzle (commercially marketed as *Hi-Q*) from the goal to the initial position appears [7] to be harder for most people than solving it from the initial to the goal state. This is surprising, since the puzzle is in principle completely reversible. Bogomolny [10] writes that

In principle, the puzzle should be as easy (or as difficult) as its original counterpart. However, I have discovered that to me, as far as the theory goes, it's much easier to visualize full [configurations] than empty ones.

As Leibniz noted [7], regressing a peg after “unjumping” feels intuitively different than updating the state after “jumping”.

The fundamental problems of forward planning for humans appear to be largely finessed by the assumptions of classical planning (pp. 2–3). In classical planning, nothing unexpected can happen during plan execution: world states persist perfectly and actions always have their desired effects. Given these simplifying assumptions, humans appear to be good forward planners, at least for real-world problems. Puzzle-mode problems tend to be hard for humans largely for reasons unrelated to temporal direction, having more to do with the large amount of short term state and the rapid search required for their solution.

The question of directional bias in human-generated encodings of STRIPS domains appears to be a largely unstudied one. This is due in part to the fact that few large real-world problem domains have ever been given STRIPS encodings. Characteristics of small or artificial problems are unlikely to generalize: the typical encoding of these problems typically addresses the qualification problem [21] by pruning the fluents involved in the problem ruthlessly. For example, neither *gratuitous* preconditions (which can be trivially satisfied in any reachable world state) nor *side effects* (unintended but unavoidable effects of an action) are normally modeled in toy problems.

One might suspect humans of routinely constructing operators with certain distinctive properties, for example:

Purpose: The purpose of an operator is a single effect that is the expected reason for insertion of the action into the plan. Humans may concentrate on the intended effect, and thus omit effects other than the purpose (the side effects described above) when constructing STRIPS operators.

Undoability: An operator is undoable if there are other operators available in the domain that can exactly undo its effects. Humans may be wary of the execution difficulties associated with actions which cannot be undone, and thus eschew these operators in constructing STRIPS problem descriptions.

Repeatability: An action is repeatable if, having executed the action, it is possible to re-establish its preconditions. Humans may include operators with perceived general utility when constructing STRIPS problem descriptions: unrepeatable operators, which can be used only once, are the essence of specificity.

This kind of reasoning is confusing, however, since humans also tend to structure the world itself so that it has these sorts of properties, in order to make planning tractable. Thus, these properties are often met by the real-world operators to which the STRIPS descriptions correspond.

In the absence of extensive study, and given the notorious unreliability of introspection, it is difficult to make strong assumptions about the role of human thought in planning. Thus, the formal results of succeeding chapters are especially important in understanding directionality in STRIPS planning.

2.1.3 The Electronic Brain: Directionality and Algorithmics

There appear to be three possible sources of directional bias in STRIPS planning. First, the STRIPS formalism itself may be easier to solve in one direction, due to the nature of persistence. Second, particular STRIPS problem domains may be easier to solve in one direction than the other. Finally, planning algorithms may choose to operate in a particular direction.

The results of chapter 3 imply that the bias in the STRIPS formalism itself is illusory. Since STRIPS problems, problem domains, and even individual actions in these domains are shown to be reversible, there cannot be any predisposition toward either forward or backward reasoning due to the formalism.

The planning problems themselves may be biased [20]. For example, by including many operators with far more effects than preconditions, problems will have a much larger forward branching factor. By leaving the goal state partly unspecified, extra branching will be introduced near the root of a backward search tree. These biases are real, and underscore the need for planners to be *bidirectional* for optimal performance. Recent research has made some progress in this area [18].

There are two general classes of STRIPS planning algorithm that typify the distinction between forward and backward planning algorithms. The first, forward state-space search, starts with the initial state and produces successor states by applying actions whose preconditions are satisfied, in an attempt to produce the goal state. One argument in favor of this approach is that since total states are being produced at each step, no complicated reasoning about what actions are legal is required for efficient pruning [3].

The second, plan-space search, is typified by “Partial-Order Causal-Link” (POCL) planning [34]. POCL planning starts with the goal state, and attempts to insert actions into a plan whose effects achieve the goal state, then actions to satisfy otherwise unsatisfiable preconditions of these actions, and so forth, until the initial state satisfies all remaining preconditions. It is this effects-to-causes means-end analysis, inserting new actions in order to supply otherwise unavailable preconditions, that leads to a characterization of POCL planning as backward.

The arguments in favor of backward POCL planning arise chiefly from an apparent reduction in the size of the search space, from two causes. First, while the initial state is almost always totally specified, it is not uncommon for only a few fluents to be specified in the goal state. Thus, the number of actions needed to achieve the goal state itself may be limited. Of course, since many

actions will have more preconditions than effects, a large number of fluents usually become involved in the plan after just a few backward steps.

Second, actions are not inserted blindly into POCL plans, but only in response to the need to achieve specific fluent values. Thus, the branching factor at each step in the planning process is lower. It is this additional reasoning about which preconditions require additional actions to be inserted into the plan that accounts for the slower search rates of POCL planners. It is hoped that the reduced search space will compensate for this slowdown. The trend away from POCL planners, however, has suggested that this is not the case.

To summarize, while there are possible sources of bias in STRIPS planning, in reality these biases appear to be either illusory or problematic. The one thing that does seem certain is that any overall algorithmic advantage to planning either forward or backward in the STRIPS formalism is uncertain and difficult to quantify.

2.2 Confounding Issues

As noted in chapter 1, even with all of the popular extensions to STRIPS there are still important concepts in planning that it fails to capture. Among these concepts are hierarchy and abstraction. Both of these concepts confuse issues of directionality by introducing approximations into the planning process that can “jump over” difficult areas. It may also be that the STRIPS formalism is not an appropriate choice for general-purpose planning: this might have implications for directionality.

2.2.1 Directionality and Hierarchy

As discussed in chapter 1, a hierarchical planning domain, problem, or planner is one in which some operators are considered subordinate to others. A typical hierarchical planner is O-Plan [16], which features a complex language for describing composite operators and specifying the fashion in which the planner can break them down and insert them into plans.

O-Plan operating on a non-hierarchical domain is a backward planner (as will be shown in chapter 5). Indeed, O-Plan eschews even partial-order techniques in favor of carefully controlled simple goal regression. But the operation of O-Plan on a hierarchical domain raises some interesting questions.

In a hierarchical problem, plans for any subgoal at any given level tend to be short: long plans are typically the result of the breakdown of many levels of hierarchical structure. Thus, a supposedly backward planner can quickly work to the beginning of a planning problem. It can then establish invariants that must hold in the initial state, and use these in planning with the next-level decomposition in the hierarchy. This mode of operation suggests forward planning: state properties are used to prune action selection on plan suffixes.

This mode of operation, however, is different from that of a typical bidirectional planner, which deliberately works from both ends of a problem. The hierarchical planner is asymmetric, and its ability to reach the front of the plan quickly is an “accident” of the hierarchy (and a computationally important one).

2.2.2 Directionality and Abstraction

Several historically important planners have been built around the idea that tractable planning can be achieved using operator or domain *abstraction*. In this approach, all but some salient features of an operator or domain are temporarily ignored, in order to get an approximate solution to the problem. This candidate solution is then refined by introducing greater and greater levels of detail. This approach is rarely used by modern planning systems, perhaps because the per-node cost of search has decreased to the point where the extra complexity of dealing with the abstractions is no longer profitable. The ABSTRIPS system [49] is perhaps the earliest planning system to incorporate abstraction as a fundamental feature. ABSTRIPS uses simple operator abstraction to iteratively refine a proposed plan.

The notion of abstraction has been used extensively in the context of *macro operators*: pre-packaged subplans that are either presented as part of the domain or, more commonly, computed at runtime. The idea behind macro operators is to avoid repeatedly rediscovering a plan for a hard subproblem. Macro operators fit nicely with operator abstraction: abstraction can produce abstract macro operators that may become more detailed as they are expanded.

As in the hierarchical case, abstraction and macro operators raise interesting questions about the directionality of planners. The ABSTRIPS planner, like the STRIPS planner, is fundamentally a forward planner. However, it is possible for ABSTRIPS to select a single operator that, at the highest level of abstraction, comprises the entire plan. Subsequent refinements of this abstract operator will add actions at arbitrary points in the plan in order to make successively more preconditions true. Thus, the forward nature of STRIPS is somewhat obscured by the ABSTRIPS abstraction.

2.2.3 The Limits of STRIPS

As noted earlier (pp. 2–3 of chapter 1), the assumptions of classical planning impose strong constraints on planning problem expressiveness, with the hope of achieving a computationally feasible version of the planning problem. The soundness of the STRIPS formalism is dependent on all of the assumptions of classical planning.

Unfortunately, it appears that it is computationally difficult to find plans even under these restrictive conditions. It is reasonable, then, to ask whether some of the classical planning constraints should be relaxed, so that formal descriptions of planning problems are at least accurate and plans obtained for them will be useful. Historically, work has been done on relaxing several of the requirements of classical planning: that the world state be freely and fully observable, by allowing *sensing actions* [31]; that the world state be complete and unchanging, by searching for *robust plans*; and that actions be atomic, through the use of temporal logic [16]. More recently, a strong interest has been taken in allowing world states to be incomplete and dynamic and actions nondeterministic, by means of *Partially Observable Markov Decision Process* planning [15].

Most of these attempts to relax the classical planning assumptions seem to have made the planning problem more difficult. In addition, none of them appear to significantly affect the issue of directionality. To the extent that the regression problem becomes harder under relaxed assumptions, backward planning will become more difficult. For example, the Zenon planner [40] permits limited regression across actions which contain preconditions including systems of linear inequalities: the machinery required for this feat is decidedly nontrivial.

2.3 Conclusion

The popular assumption in the planning literature that backward planners have an inherent advantage in general use appears to have a rather weak foundation. Because the STRIPS formalism is not expressive enough to capture physical laws directly, physical notions of entropy do not introduce a temporal arrow into the formalism.

Similarly, human planning appears to be performed in different temporal directions in different situations. It does appear that in toy problems of the sort often used as examples in AI, humans tend to reason backward: this may say more about the problems themselves than about any general or inherent advantage of backward planning.

Finally, the computational properties of the STRIPS formalism do not appear to be directionally biased. While real-world problems may favor backward chaining, the author is unaware of any detailed study of this assertion. Chapter 3 will show that forward and backward chaining can be made equivalent in a strong sense in the STRIPS formalism. Most planning algorithms proposed recently are not of the POCL variety, perhaps as a result of the limitations of this approach.

Any philosophical discussion of directionality in planning must be constrained by the concepts and mechanisms available. A wide range of techniques have been brought to bear on STRIPS planning, but none of them appear to strongly favor backward planning. The STRIPS encoding itself may be inadequate, although no particularly attractive substitute has been proposed. There is no reason to believe that more expressive encodings of the planning problem will make backward planning more attractive; they may make it less so.

Directional arrows in planning are scarce, and most are confusing. Neither the physical world, the mental world, nor the nature of the formalism provides an unambiguous clue. Better understanding can be obtained only through detailed theoretical and experimental study of the sort undertaken in the remainder of this work.

Chapter 3

STRIPS Problem Reversal

It has been assumed (as discussed on pp. 12–13) that there is an inherent directional asymmetry in the STRIPS formalism, independent of particular problems or domains, arising from the persistence of fluent values central to STRIPS. This asymmetry is largely illusory: directional asymmetry in STRIPS planning arises either from the specific planning problem or from a specific choice of planning algorithm, rather than from the STRIPS formalism itself.

To show this, a simple, tractable construction is given, based on a technique for domain compilation. For any PROPS problem P , the construction produces a problem P_R with an isomorphic set of operators and fluents, such that the set of reversals of plans for P_R is exactly isomorphic to the set of plans for P . Because of the strong correspondence between the original and reverse domains, it is possible to treat forward planning in P as backward planning in P_R , and vice-versa. This technique extends to PREDS, and can be used to allow unidirectional planners to plan in the direction opposite their natural direction.

Methods of transforming operators have been explored previously, for example by Gazen and Knoblock [19] to simplify PREDS domains for use with Graphplan. The methods of operator transformation given in this chapter, however, are distinguished from previous work by their goal of controlling search and by their extensive nature: the compilation schema given here rewrite every operator in a problem description.

3.1 PROPS Actions and the Frame Axiom

As discussed previously, the principal advantage claimed for STRIPS formalisms over their earlier predecessors requiring general-purpose theorem proving is the incorporation of persistence information into the STRIPS formalism. This relieves the planner of the burden of making persistence-based inferences using a general purpose mechanism. As a corollary, it has long been assumed that the persistence of values into the future, but not into the past, creates causal and/or temporal asymmetry in STRIPS planning, so that backward planners, which work from the goal conditions of a planning problem toward the initial conditions, must of necessity take a different approach than forward planners, which work from the initial conditions toward the goals.

This assumption has arisen from the view of means-end analysis as central to STRIPS planning. In the traditional view of means-end analysis (chapter 1 p. 3), one fixes an end, namely a fluent value to be achieved at a particular point in the plan, a means, namely an earlier action that achieves it, and

then ensures that intervening actions in the plan do not interfere with the persistence of the achieved value. This sort of reasoning was central to the operation of the original STRIPS [17] system itself (although in the initial version of this system the non-interference of intermediate actions with fluent persistence was *assumed*). More recently, the above definition can be seen to have led directly to the notion of POCL planning.

A careful examination of PROPS actions, however, suggests that these actions are actually almost time-symmetric: exchanging the preconditions and effects appears to be largely locally sound. Persistence of effects thus appears as persistence of preconditions (i.e., an action's preconditions persist up until the action that produced them).

This observation leads to the notion of *reversal* of a PROPS planning problem: exchanging the preconditions and effects of each action, and exchanging the initial and goal conditions, in order to obtain a problem isomorphic to the original, but whose plans are the reversal of plans in the original problem.

3.1.1 Reversibility and Deleted Preconditions

Perhaps the simplest planning algorithm is forward state-space search, in which plans are built from the initial state by repeatedly appending actions. Note that to produce a total state as a result of an action, one must apparently know the total state preceding the action, since otherwise one cannot know the values of fluents unaffected by the action.

Superficially, it would seem straightforward to produce the total state *preceding* an action by knowing the total state *after* the action (i.e., solve the regression problem of chapter 2), but two factors intervene here. First, it is common in STRIPS problem descriptions to give a partial goal description rather than a total goal state. Second, and more importantly, STRIPS actions can set the value of fluents not mentioned in their preconditions, resulting in the inability of a backward planner to assume a fluent value before an action is taken. Consider a fluent from a cooking problem that represents whether a spoon is wet, and the action of stirring a liquid. A forward planner can infer that the spoon should be wet after the liquid is stirred, but a backward planner cannot determine, solely from the succeeding state and the formalism, the wetness of the spoon in the preceding state. This action thus appears to be irreversible.¹

Total state-space search thus appears to require proceeding forward from the initial state.² In the next section, however, a mechanism is given for reversing these “irreversible” operators, allowing total state-space search in either direction.

3.2 Reversal Using Compilation

It is reasonable to view the process of converting one PROPS domain or problem to another as a *domain compilation* process that takes actions and fluents in the source domain and translates them to isomorphic actions and fluents in the target domain. This section gives several examples of

¹This loss of information has been equated with *entropy* in physical systems, with the application of an “irreversible” planning operator regarded as “increasing entropy.” See p. 13 of chapter 2 for a detailed discussion of this view.

²Some PREDS domains do contain only *reversible* operators, in which all fluents mentioned in effects are also mentioned in preconditions. This appears to be particularly true in toy domains, where actions are carefully crafted to have particular roles in plans.

such compilation schema, culminating in a domain compilation scheme C_R that reverses a PROPS domain. This is done by transforming the fluent space and the operators slightly, in such a way that even “irreversible” operators may be reversed, and the initial and goal states may be interchanged (potentially producing a partially-specified initial state). Every operator in P has a corresponding operator, its reversal in P_R . Every fluent in P corresponds to a pair of fluents in P_R (whose values change in lock-step). Thus, PROPS planning algorithms that reason forward in P_R are effectively reasoning backward in P , and algorithms that reason backward in P_R are effectively reasoning forward in P .

3.2.1 C_2 : Positive-Only Preconditions

It is not uncommon in PROPS to restrict the actions in a planning domain to allow only positive preconditions. It is well known that this does not lead to a restriction on the expressive power of planning domains. The reason for this is simple. One can always force each action to maintain two versions of each original fluent f : f^+ which is true when f would be true, and f^- which is true when f would be false. Thus, each action that adds f^+ in an effect must delete f^- , and vice-versa.

Definition 3.1 (Signed Fluents) *The set of signed fluents corresponding to a set of fluents F is given by*

$$F^\pm = \{f^+, f^- \mid f \in F\}$$

Figure 3.1 shows the compilation rules for a compilation scheme C_2 that converts a planning domain with arbitrary preconditions to one with only positive preconditions: for each possible way in which a fluent f can appear in an action in the source domain, there exists a C_2 rule that shows how to render f in terms of f^+ and f^- in the compiled domain.

The compilation rules for a PROPS domain D may be extended to compilation rules for a PROPS problem P over D in the obvious fashion, by simply transforming the fluents in the initial and goals states according to the domain compilation rule.

Definition 3.2 (Signed Formula) *For any conjunctive formula F (definition 1.3), the signed formula F^s is given by*

$$F^s = \{f^+, \neg f^- \mid f \in F\} \cup \{\neg f^+, f^- \mid \neg f \in F\}$$

This permits a precise description of problem compilation: For a PROPS problem

$$P = \langle \langle F, A \rangle, I, G \rangle$$

the compiled problem is given by

$$C_2(P) = \langle \langle F^\pm, C_2(A) \rangle, I^s, G^s \rangle$$

Note that some combinations of fluents in the source domain have been omitted since they can be obtained from existing combinations by *persistence*: for example, the appearance of a fluent f in an action as $\overset{f}{-}$ is indistinguishable from a planning point of view from its appearance as $\underset{f}{f}$. This explicit persistence will be useful in the proofs that follow.

$$\frac{f}{f} \Rightarrow \frac{f^+}{f^+ \neg f^-} \quad (3.1)$$

$$\frac{\neg f}{\neg f} \Rightarrow \frac{f^-}{\neg f^+ f^-} \quad (3.2)$$

$$\frac{f}{\neg f} \Rightarrow \frac{f^+}{\neg f^+ f^-} \quad (3.3)$$

$$\frac{\neg f}{f} \Rightarrow \frac{f^-}{f^+ \neg f^-} \quad (3.4)$$

$$\overline{\frac{f}{f}} \Rightarrow \overline{\frac{f^+}{f^+ \neg f^-}} \quad (3.5)$$

$$\overline{\frac{\neg f}{\neg f}} \Rightarrow \overline{\frac{f^-}{\neg f^+ f^-}} \quad (3.6)$$

Figure 3.1: Scheme C_2 : Compilation to Positive-Only Preconditions

To compile an action, one compiles each of its fluents separately. For example

$$\frac{a \neg b}{b \neg c}$$

would first be transformed to make the persistence of a explicit, yielding

$$\frac{a \neg b}{a b \neg c}$$

and then would compile according to the C_2 rules into

$$\frac{a^+ b^-}{a^+ \neg a^- b^+ \neg b^- \neg c^+ c^-}$$

One further subtlety that might not be immediately apparent is that some conditions are explicitly reasserted in effects that must, at this stage, already implicitly hold in the preceding state (as a consequence of the domain structure). For example, consider rule 3.1 in figure 3.1. It is apparent that f^- must not hold in any state legally preceding an action of this type in the compiled domain, so it is redundant to delete f^- as part of the effect. For reasons that should become clear shortly, it is nonetheless convenient to do so.

It is, of course, necessary to prove that a compilation scheme does not change the problem.

Definition 3.3 (Compilation Correctness) *A compilation scheme for a PROPS problem is complete if every valid plan for the source domain, when compiled, is a valid plan for the compiled domain. A compilation scheme is sound if every valid plan in the compiled domain can be produced by compiling a valid plan in the source domain. A compilation scheme is correct if it is complete and sound.*

Thus, for a complete compilation scheme no existing plans are eliminated in the compiled problem, and for a sound compilation scheme no new plans are allowed in the compiled problem. The technique used below for proving that C_2 is complete and sound is somewhat unusual, but is justified by the fact that this proof structure will be used repeatedly.

The basic idea behind the proof of correctness for C_2 is to compare the goal state and all preconditions of each action in the source problem with the corresponding formulae in the compiled problem. Essentially, the proof proceeds by showing that, for every valid plan in the source problem, each precondition and the goal state are satisfied in the corresponding plan in the compiled problem (completeness). For every valid plan in the compiled problem, it is then shown that each precondition and the goal state are satisfied in the source problem (soundness).

To achieve these proofs, it is sufficient to note that, because of the way that persistence across actions is eliminated by the encoding, a precondition can only be satisfied if (1) there is an earlier effect that establishes that precondition and (2) no action between the establishing effect and the precondition mentions the precondition fluent. (Any fluent mentioned by an action in the normalized form is part of the effect of that action. This implies that the establishing action is unique.) Thus, it is sufficient to compare compilation rules in the source and compiled domains in a pairwise fashion, checking that the set of legal action pairs in each domain is exactly the same.

Table 3.1: Possible a_0, a_1 (\hat{a}_0, \hat{a}_1) Pairs for C_2

a_0 (\hat{a}_0)	a_1 (\hat{a}_1)
3.1	3.1, 3.3, 3.5, 3.6
3.2	3.2, 3.4, 3.5, 3.6
3.3	3.2, 3.4, 3.5, 3.6
3.4	3.1, 3.3, 3.5, 3.6
3.5	3.1, 3.3, 3.5, 3.6
3.6	3.2, 3.4, 3.5, 3.6

Proposition 3.1 C_2 is correct.

PROOF: Completeness and soundness are argued separately.

Completeness:

Without loss of generality, consider any valid plan for the source problem and some action a_1 in this plan which mentions a fluent f (and thus affects f due to the normalization described above). Since a_1 is part of a valid plan, its precondition on f (if any) must be satisfied. This can happen in one of two ways.

1. f may have the same sign in the initial state as in the precondition of a_1 with no action preceding a_1 mentioning f .
2. Otherwise, there must be some action a_0 preceding a_1 such that a_0 adds f with the appropriate sign and no action between a_0 and a_1 mentions f .

Now consider the corresponding sequence of actions in the compiled problem, and in particular the compiled action \hat{a}_1 . If a_1 mentioned f in its preconditions, \hat{a}_1 must require exactly one of f^+ or f^- . To see that any precondition \hat{f} of \hat{a}_1 is satisfied, consider the two previous cases:

1. The initial state I provides the precondition f of a_1 if and only if \hat{f} appears with the appropriate sign in \hat{I} to satisfy \hat{a}_1 .
2. If action a_0 provided the precondition f of a_1 , then this places a restriction on how a_0 and a_1 can treat f .

Table 3.1 enumerates all the possible ways in which a_1 can follow a_0 , assuming that both a_0 and a_1 mention f , and no intervening action does. The left column of table 3.1 indicates the rule in figure 3.1 which matches a_0 on f , and the right column indicates the rule which matches a_1 . Table 3.1 also shows the ways that \hat{a}_0 can supply f^+ or f^- to \hat{a}_1 with the appropriate sign: a relation identical to that in the source domain.

(For example, if $a_0 = \frac{f}{f}$ in the source domain, the mention of f in a_1 might be of the form

$\frac{f}{\neg f}$. In the compiled domain, $\hat{a}_0 = \frac{f^+}{f^+ \neg f^-}$ and $\hat{a}_1 = \frac{f^+}{\neg f^+ f^-}$. Thus, \hat{a}_1 can follow \hat{a}_0 in this instance.)

Note that the argument above also applies to any goal fluent f , so all goal fluents will be satisfied in the compiled problem if they are satisfied in the source problem. Thus, since every precondition of every action and every goal fluent in any valid plan will be satisfied in the compiled problem if it is satisfied in the source problem, the compilation scheme C_2 is correct.

Soundness:

The soundness argument is similar to the completeness argument. Without loss of generality, consider any valid plan for the compiled problem and some action \hat{a}_1 in this plan which mentions a fluent f^+ or f^- . Since \hat{a}_1 is part of a valid plan, its precondition on f^+ or f^- (if any) must be satisfied. By symmetry, assume without loss of generality that the precondition is f^+ . This precondition can be satisfied in one of two ways.

1. f^+ may occur in the initial state, with no action preceding \hat{a}_1 mentioning f^+ .
2. Otherwise, there must be some action \hat{a}_0 preceding \hat{a}_1 such that \hat{a}_0 adds f^+ and no action between \hat{a}_0 and \hat{a}_1 mentions f^+ .

Now consider the corresponding sequence of actions in the source problem, and in particular the source action a_1 . If \hat{a}_1 mentioned f^+ in its preconditions, a_1 must require f as a precondition. To see that any precondition f of a_1 is satisfied, consider the two previous cases:

1. The compiled initial state I provides the precondition f^+ of \hat{a}_1 if and only if f appears positively in I to satisfy a_1 .
2. If action \hat{a}_0 provided the precondition f^+ of \hat{a}_1 , then this places a restriction on how \hat{a}_0 and \hat{a}_1 can treat f . Table 3.1 shows that the ways that a_1 can follow a_0 in the source domain are identical, and thus a_0 supplies f to a_1 .

Note that the argument above also applies to any goal fluent f^+ or f^- , so all goal fluents will be satisfied in the source problem if they are satisfied in the compiled problem. Thus, since every precondition of every action and every goal fluent in any valid plan will be satisfied in the source problem if it is satisfied in the compiled problem, the compilation scheme C_2 is sound.

3.2.2 C_3 : “Don’t Care” Effects

One can define a new type of effect for PROPS-like actions: a “Don’t Care” or DC effect. That a fluent f is DC will be indicated by writing it as $*f$: the definition of effects of actions may be extended to include atomic formulae of this type. Semantically, a DC effect $*f$ puts a fluent f in a state that allows it to satisfy either positive or negative preconditions on f . For example, both

$$\frac{f}{*f} \quad \frac{f}{-}$$

and

$$\frac{f}{*f} \quad \frac{-f}{-}$$

$$\frac{f}{*f} \Rightarrow \frac{f^+}{f^+ f^-} \quad (3.7)$$

$$\frac{\neg f}{*f} \Rightarrow \frac{f^-}{f^+ f^-} \quad (3.8)$$

Figure 3.2: Scheme C_3 : Extending C_2 to 3-Valued Logic

are valid plans. For now, the use of DC effects will be restricted to fluents mentioned in the precondition of an action. Thus

$$\overline{*f}$$

is not a valid action.

It might be supposed that the introduction of restricted DC effects increases the expressive power of PROPS. It turns out, however, that this is not the case: A simple extension to C_2 produces a new compilation scheme C_3 that translates a PROPS domain with restricted DC effects to a standard PROPS domain (with positive preconditions only, as an added bonus). Figure 3.2 shows the extra compilation rules for PROPS with restricted DC effects. For a PROPS problem

$$P = \langle \langle F, A \rangle, I, G \rangle$$

the compiled problem is given by

$$C_3(P) = \langle \langle F^\pm, C_3(A) \rangle, I^s, G^s \rangle$$

As before, a proof of the correctness of this compilation scheme is required. The proof technique mirrors that for C_2 ; the principal difference is that the set of legal pairs changes.

Proposition 3.2 C_3 is correct.

PROOF:

The proof structure is as before. Now, however, the table of possible action pairs contains some extra entries, as shown in table 3.2.

3.2.3 C_R : Reversal

For the moment, consider the restricted class of PROPS problems with a total initial and goal state (both I and G mention all fluents). The compilation scheme C_R of figure 3.3 provides a way to *reverse* the PROPS problem. Given a PROPS problem

$$P = \langle \langle F, A \rangle, I, G \rangle$$

Table 3.2: Possible Pairs for C_3

$a_0 (\hat{a}_0)$	$a_1 (\hat{a}_1)$
3.1	3.1, 3.3, 3.5, 3.6, 3.7
3.2	3.2, 3.4, 3.5, 3.6, 3.8
3.3	3.2, 3.4, 3.5, 3.6, 3.8
3.4	3.1, 3.3, 3.5, 3.6, 3.7
3.5	3.1, 3.3, 3.5, 3.6, 3.7
3.6	3.2, 3.4, 3.5, 3.6, 3.8
3.7	3.1–3.8
3.8	3.1–3.8

$$\frac{f}{f} \xLeftrightarrow{\quad} \frac{f}{f} \quad (3.9)$$

$$\frac{\neg f}{\neg f} \xLeftrightarrow{\quad} \frac{\neg f}{\neg f} \quad (3.10)$$

$$\frac{f}{\neg f} \xLeftrightarrow{\quad} \frac{\neg f}{f} \quad (3.11)$$

$$\frac{\neg f}{f} \xLeftrightarrow{\quad} \frac{f}{\neg f} \quad (3.12)$$

$$\frac{-}{f} \xLeftrightarrow{\quad} \frac{f}{*f} \quad (3.13)$$

$$\frac{-}{\neg f} \xLeftrightarrow{\quad} \frac{\neg f}{*f} \quad (3.14)$$

Figure 3.3: Scheme C_R : Reversal Rules

the reversal is given by

$$C_R(P) = \langle \langle F, C_R(A) \rangle, G, I \rangle$$

Let ρ be a valid plan in the source problem, and $R(\rho)$ be the reversal of ρ . Similarly, let ρ_R be a valid plan in the compiled problem, and $R(\rho_R)$ its reversal. To see that C_R is correct, it is sufficient to establish that (1) for any plan ρ for the source problem, $R(\rho)$ is valid in the compiled problem, and that (2) for any valid plan ρ_R for the compiled problem, $R(\rho_R)$ is valid in the source problem. This can be accomplished by once again showing that all preconditions and goal conditions in one plan are satisfied by isomorphic actions in the other.

Proposition 3.3 C_R is correct.

PROOF: *Essentially, the structure of this proof is the same as that of the proofs of propositions 3.1 and 3.2, except that preconditions in the source problem correspond to effects in the compiled problem, and vice versa.*

Completeness:

Without loss of generality, consider any valid plan in the source problem, and some effect upon a fluent f . There are two possibilities for how this effect can be produced:

1. f is produced by the initial state.
2. f is produced by some action a_0 .

There are also two possibilities for how the effect can be used:

1. The next action a_1 that mentions f must either not have a precondition that mentions f or must mention f with the correct sign.
2. If there are no further actions mentioning f , f must have the correct sign in the goal state.

The interesting case is when both a_0 and a_1 exist. In this case the precondition on f required by a_1 (if any) is supplied by a_0 in the source problem. Thus, when the problem is reversed, an effect of \hat{a}_1 must supply the precondition f for \hat{a}_0 . Table 3.3 enumerates all possible ways in which a_0 can precede a_1 if a_0 has an effect on f . Table 3.4 enumerates all possible ways in which \hat{a}_0 can precede \hat{a}_1 if \hat{a}_0 has an effect on f . Note that the relations defined by tables 3.3 and 3.4 are the inverse of one another: thus, action a_0 can supply a precondition on f for action a_1 in the source problem if and only if action \hat{a}_1 can supply the same precondition to \hat{a}_0 in the compiled problem. Thus the reversed plan is valid.

In the case that a precondition in the source problem is supplied by the initial state, this corresponds to an effect in the goal state in the compiled problem. Thus, since the initial state is used to satisfy the precondition f of a_1 in the source problem, in the compiled problem the goal condition f is produced by the compiled action \hat{a}_1 . Initial conditions in the source problem not affected by any action must also be true in the goal states: the reversal of the initial and goal states will not change this. Thus, the goal state is achieved in the compiled problem.

Since a valid plan in the source problem is also valid in the compiled problem, C_R is sound.

Table 3.3: Possible a_0, a_1 Pairs for C_R

a_0	a_1
3.9	3.9, 3.11, 3.13, 3.14
3.10	3.10, 3.12, 3.13, 3.14
3.11	3.10, 3.12, 3.13, 3.14
3.12	3.9, 3.11, 3.13, 3.14
3.13	3.9, 3.11, 3.13, 3.14
3.14	3.10, 3.12, 3.13, 3.14

Table 3.4: Possible \hat{a}_0, \hat{a}_1 Pairs for C_R

\hat{a}_0	\hat{a}_1
3.9	3.9, 3.12, 3.13
3.10	3.10, 3.11, 3.14
3.11	3.9, 3.12, 3.13
3.12	3.10, 3.11, 3.14
3.13	3.9–3.14
3.14	3.9–3.14

Soundness:

The completeness proof for C_R mirrors the structure of the soundness proof, except it shows that any valid plan in the compiled problem is also a valid plan in the source problem.

The only unique part of this proof is showing that for any compiled action \hat{a}_1 that can precede \hat{a}_0 with respect to f , a_1 can follow a_0 in the source problem. Again, tables 3.3 and 3.4 show this.

Thus, C_R is sound and complete, and therefore correct.

Note the role that the DC effects play in C_R . Intuitively, the reason that the rules 3.13 and 3.14 of C_R work is that, when looking backward, the decision as to whether f should be positive or negative may be deferred until it is known which sign of f is needed to satisfy the precondition of an action. The compilation scheme C_3 thus plays a critical role, in that it allows standard PROPS planning domains to represent this DC effect using Boolean fluent values.

To reverse a PROPS problem P is a two-step process. First, compilation scheme C_R is applied to produce a PROPS problem $C_R(P)$ that may contain actions with DC effects. Then C_3 is applied to remove any DC effects, converting the description into a problem $P_R = C_3(C_R(P))$ in standard PROPS formalism. The resulting problem has the property that forward plans for P are backward plans for P_R and vice-versa. Since each action is treated singly and separately, and the number of fluents at most doubles, the compilation takes at most linear time in either of these measures.

3.2.4 Partial Goal States and Reversal

It is not uncommon for a PROPS problem to have a goal description corresponding to a partial, rather than total, goal state. The reversal technique is easily extended to apply to such a problem, by making the goal state total during compilation. Essentially, unspecified goal fluents in the source problem P will correspond to DC initial fluent values in P_R . The following definition formalizes this process.

Definition 3.4 (State Expansion) *The state expansion $\text{expand}(G, F)$ to fluents F of a (partial) state $G \subseteq F$ consists of the fluents of G , together with DC fluents for each of the remaining fluents in F .*

$$\text{expand}(G, F) = G \cup \{ *f \mid f \in (F \setminus G) \}$$

The compilation of a DC fluent in a state can be given by an extension of the definition of signed formula (definition 3.2).

Definition 3.5 (Signed Formula) *For any conjunctive formula F possibly containing DC fluents, the signed formula F^s is given by*

$$F^s = \{ f^+, \neg f^- \mid f \in F \} \cup \{ \neg f^+, f^- \mid \neg f \in F \} \cup \{ f^+, f^- \mid *f \in F \}$$

To reverse a PROPS problem P with partial goal states, it is necessary to expand the goal state during reversal. Given a PROPS problem $P = \langle \langle F, A \rangle, I, G \rangle$ where G may be a partial goal description, the reversal is given by

$$C_r(P) = \langle \langle F, C_r(A) \rangle, \text{expand}(G, F), I \rangle$$

Given the same semantics for DC fluents in the initial state as for DC effects, the proof of correctness for C_r (proposition 3.3) can be seen to cover C_r as well.

To produce a standard PROPS version of P_R where the goal description $G(P)$ may be partial is again a two-step process. First, compilation scheme C_r is applied, producing a PROPS problem $C_r(P)$ that may contain actions with DC effects, and may contain DC fluents in the initial conditions. Then C_3 is applied (using the new definition of signed fluent to translate DC fluents) to remove any DC effects or fluents, converting the description into a problem $P_R = C_3(C_r(P))$ in standard PROPS formalism.

3.3 Extensions and Reversibility

As discussed earlier, a number of extensions to PROPS have been proposed. It is important to consider which of these are compatible with the reversal technique, in order to gauge its generality and applicability. PREDS and the extensions described in chapter 1 (pp. 9–10) are considered in turn:

Predicate Strips: Inspection of the reversal proof indicates that it lifts easily to the predicate case: if the ground atomic conjunctive formulae of the proof are replaced by conjunctive formulae over universally-quantified predicates, the formalism is unchanged. Thus, one can reverse Predicate STRIPS planning domains as well as PROPS domains.

Closed World Assumption: The presence or absence of the CWA, in either its PROPS or PREDS formulations, does not affect the reversal mechanism, since the mechanism is dependent only on fluent values, not on what fluents are available. In PREDS formulations in which the fluents are not explicit, they will have to be inferred as needed during compilation: whatever mechanism the planner normally uses for this should suffice.

Types: Any type assignment applied to the fluents of the original problem will transfer in a straightforward fashion to the fluents of the reversed problem. Since type predicates are timeless, they need not be compiled at all for reversal purposes.

Conditional Effects: Reversing actions with conditional effects (presumably producing reversed actions with conditional effects) is problematic. It is possible that the conditional portion of the action can be reversed as though it is a complete action. However, the formalism used in the chapter is not quite sufficient for describing this, and in any case it is not obvious that interactions between the absolute and conditional preconditions of the action will not adversely affect the reversal. Thus, at present the reversal question for conditional effects is still open: this is an important topic for future work.

Domain Axioms: Domains containing domain axioms are not reversible by compilation. In the presence of domain axioms, forward and backward search really can be fundamentally different: the domain axioms themselves can introduce a temporal arrow into the problem. For example, given a sufficiently powerful formalism for domain axioms, it is possible to specify a one-way function (see chapter 4) of a set of fluents as a domain axiom: this will force the problem to be tractable only in one direction. On the other hand, given the semantics and tractability problems associated with domain axioms, this is arguably not a significant limitation of the reversal technique in practice.

Safety Constraints: Because safety constraints are imposed on states rather than actions, no special mechanism is required to impose identical safety constraints in a reversed domain.

There is, however, one catch. The Weld and Etzioni formulation of safety constraints specifies that a safety constraint that is violated in the initial conditions may remain violated [58]. The purpose of this odd-looking proviso is to sidestep the undecidability of mutual consistency of first-order safety constraints. Unfortunately, permitting actions which maintain a safety constraint violation in the forward direction does not have any obvious backward analogue. An alternative would be to insist on some restricted form of safety constraints whose mutual satisfiability is decidable. Indeed, most natural safety constraints seem to be expressible as Horn clauses.

Overall, the reversal mechanism appears to be sufficiently general to handle interesting cases. The general approach behind it may be extensible even when the specific technique is not.

3.4 Conclusions

If it is sensible to talk about the reversal of STRIPS actions, domains, or problems, it is probably no longer sensible to talk about the directionality of STRIPS actions, domains, or problems. The

temporal directionality provided by persistence appears to be illusory; an action's preconditions are time-symmetric with its effects. Any time-asymmetric action can be replaced with an isomorphic action that is time-symmetric. At the least, this suggests that any temporal asymmetry in STRIPS problems arises from the problems themselves, rather than the STRIPS encoding.

The use of domain compilation to provide a simple, tractable STRIPS problem reversal algorithm has several consequences. First, it implies that a common belief about the nature of the STRIPS formalism, namely that persistence induces a temporal arrow in this formalism, is mistaken. Second, it provides a way to squeeze extra performance out of existing PREDS planning systems, by allowing them to reverse problems as needed to deal with problem-specific directional biases. Finally, it illustrates the utility of the domain compilation technique for providing more powerful planning operators.

Chapter 4

Determining The Directionality Of Planners

When considering the directionality of planning algorithms, several questions arise:

1. How should the directionality of a planning algorithm be *defined*?
2. How can the directionality of a planning algorithm as defined in question (1) be *determined*?
3. How does the directionality test of question (2) work out in practice?

This chapter answers question (1) by giving an intuitively acceptable formal definition of planning direction, and addresses question (2) by giving a powerful technique for determining planner direction that is provably correct even for poorly-understood STRIPS-style planners of sufficient power. Question (3) is answered in chapter 5, where a modification of the technique of this chapter is applied experimentally in order to better understand the performance of several planners.

The focus of this chapter is on an extrinsic, or black-box,¹ technique for determining the directionality of a STRIPS planner. This technique operates by feeding the planner problems drawn from a class of artificial problem domains, and measuring planner performance. This idea of constructing artificial problems to determine planner behavior is not new [5]. However, by exploiting the properties of certain functions (cryptographic one-way functions), it is possible to greatly strengthen both the meaningfulness of and the confidence in the results of this technique.

This chapter restricts itself to a study of directionality in PROPS. Since PREDS planners are capable of operating on PROPS domains, this restriction does not lose generality, and the extra simplicity greatly eases the construction and analysis. Further, this approach is well suited to planners such as `blackbox` [29] that are fundamentally propositional in nature.

4.1 Search Space Planning

The first consideration is to develop a general model of the action of planners, in order to provide an intrinsic definition of forward and backward planning. It is important, however, not to over-generalize: as discussed previously (chapter 2, pp. 16–17), the issue of search direction really only

¹Not to be confused with the `blackbox` planner discussed below.

applies to planners that operate by searching. Thus, a model, such as Kambhampati’s Refinement Planning [27], that encompasses arbitrary planners, may be inappropriately broad for discussion of planning direction.

The model presented here attempts to capture the mechanism common to search-based planners. The basic idea behind the model is the observation that search in planning is of two types: action selection and action ordering. In a typical planner these are interleaved, and thus cannot be treated entirely separately. A search-based planner searches in a state space consisting of partially-formulated plans, in which some actions have been selected and some ordering decisions have been made. Because of the potential terminological confusion with POCL planning and the like, a fresh terminology has been adopted: these partially formulated plans are referred to as *approaches*.

Definition 4.1 (Approach and Development) *An approach $\sigma(P)$ to a planning problem P is a tuple $\langle A, \Delta \rangle$, where A is a set of action instances drawn from P and Δ is a partial order upon them. The problem P will be dropped where it is not required to avoid ambiguity.*

A development of an approach σ is an approach σ' such that $A(\sigma) \subseteq A(\sigma')$ and $\Delta(\sigma) \subseteq \Delta(\sigma')$. A development is nontrivial if $A(\sigma) \neq A(\sigma')$ or $\Delta(\sigma) \neq \Delta(\sigma')$.

An approach is more general than a POCL partial plan: there is no notion that in the definition of approach that corresponds to partial links, or indeed to any kind of goal protection.

Developments of an approach correspond to action selection and action ordering. Any planner that explores the space of developments starting from the empty approach is a search space planner.

Definition 4.2 (Search Space Planner) *A search space planner performs combinatorial search in a (explicit or implicit) space whose nodes are approaches. The root node is the empty approach. Each child of a node σ is a legal development σ' of σ . At the leaves, Δ is a total order, and the ordered set of actions A is a linear plan.*

Note that not all of these leaf linear plans are necessarily legal. An important distinction, though one not explored in this work, is between planners that traverse only legal nodes in the search space (as state-space planners do) and those that allow arbitrary exploration. Nothing in the definitions so far constrains a planner to select and order actions in any particular way: a search space planner could in principle proceed by iterative sampling, making repeated random walks through the search space.

4.1.1 Search Direction

In order to talk about search direction, it is necessary to further specify the planning algorithm. The key intuition here is that planner direction is given by the method used for action selection. Consider a planner developing an approach by adding a new action.

Definition 4.3 (New Actions) *For a development σ' of an approach σ , the set of new actions $A^+(\sigma, \sigma')$ is defined by $A^+(\sigma, \sigma') = A(\sigma') \setminus A(\sigma)$*

A planner searches forward when it attempts to append a new action to existing actions in an approach, and backward when it attempts to prepend a new action to existing actions. A planner that searches only backward is a strongly backward planner, and a planner that searches only forward is

a strongly forward planner. An action added by a directional search step may be such that no legal plan can possibly arise: this is irrelevant. What is relevant is that the action is added because of the shape of actions added earlier or later.

For each approach explored by a strongly forward planner, only developments of that approach in which all new actions could plausibly be appended to existing actions are explored.

Definition 4.4 (Strongly Forward Planner) *The established conditions of a set of actions is the union of all the action effects (note that this may not be a consistent set).*

$$\text{est}(A) = \bigcup_{a \in A} \text{eff}(a)$$

The established conditions of an approach are the effects of all actions in the approach, together with the initial conditions: $\text{est}(\sigma(P)) = I(P) \cup \text{est}(A(\sigma(P)))$. An action satisfies the forward chaining property for an approach when it requires an established condition of that approach, or when it has no preconditions.

$$\text{fwc}(\sigma(P)) = \{a \in A(P) \mid \text{est}(\sigma(P)) \cap \text{pre}(a) \neq \emptyset \vee \text{pre}(a) = \emptyset\}$$

A development σ' in which all new actions satisfy the forward chaining property

$$A^+(\sigma, \sigma') \subseteq \text{fwc}(\sigma)$$

is a forward search step.

A strongly forward planner is a search space planner in which a development of an approach is constructed only if it is a forward search step.

As an example, consider a forward state space planner (such as ASP [11]). In a forward state space planner, the search space consists of developments of the empty approach in which the partial order Δ is actually a total order on the actions A of an approach. Each new action added to an approach during development must have *all* of its initial conditions supplied by established fluents of the approach, since it must be legal in the state produced by the current plan prefix. Thus, a forward state space planner is indeed strongly forward according to definition 4.4.

For each approach explored by a strongly backward planner, only developments of that approach in which all new actions could plausibly be prepended to existing actions are explored.

Definition 4.5 (Strongly Backward Planner) *The required conditions of a set of actions is the union of all the action preconditions (note that this may not be a consistent set).*

$$\text{req}(A) = \bigcup_{a \in A} \text{pre}(a)$$

The required conditions of an approach are the preconditions of all actions in the approach, together with the goal conditions: $\text{req}(\sigma(P)) = G(P) \cup \text{req}(A(\sigma(P)))$. An action satisfies the backward chaining property for an approach when it provides a required condition of that approach, or when it has no effects (is a no-op).

$$\text{bwc}(\sigma(P)) = \{a \in A(P) \mid \text{req}(\sigma(P)) \cap \text{eff}(a) \neq \emptyset \vee \text{eff}(a) = \emptyset\}$$

A development σ' in which all new actions satisfy the backward chaining property

$$A^+(\sigma, \sigma') \subseteq \text{bwc}(\sigma)$$

is a backward search step.

A strongly backward planner is a search space planner in which a development of an approach is constructed only if it is a backward search step.

As an example, consider a POCL planner (such as UCPOP [41]). In a POCL planner, each new action is added to a partial plan only to satisfy an *open condition* of the partial plan. The action thus supplies a required fluent of the partial plan, since it satisfies the open condition. Thus, a POCL planner is indeed strongly backward according to definition 4.5.

4.1.2 Propagation

One might reasonably object that definitions 4.4 and 4.5 are somewhat restrictive. Consider a situation in which some action a is provably part of *any* plan for some planning problem P . This might be for some simple reason. For example, a might be the only action that can achieve some goal condition: in this case, it seems unreasonable to say that an otherwise forward planner could not add a immediately. Similarly, a might be the only action executable in the initial state: it seems unreasonable to disallow an otherwise backward planner immediately adding a to its development.

To allow for these sorts of cases, it is useful to relax the strong definitions a bit. What is wanted is some notion of the immediate addition to an approach of actions that are logically entailed by that approach, and can be simply proven so: a notion generally known as *propagation*. Unfortunately, the notion of simple proofs of action entailment is hard to pin down. Fortunately, it is in any case rather peripheral to the issues at hand: the definitions of action propagation given here are sufficiently general to cover the obvious case, and more general definitions are unlikely to affect the proofs of this chapter.

The propagation definition given here requires identifying that portion of an approach that is a linear plan and whose actions occur at the beginning of the approach: this establishes a state from which forward propagation can proceed.

Definition 4.6 (Plan Prefix) Given an approach σ to a planning problem with initial state I , let $\rho_{pre}(\sigma) = \langle a_1, \dots, a_n \rangle$ be the longest sequence of actions such that

1. $a_1 \dots a_n \in A(\sigma)$
2. $\rho_{pre}(\sigma)$ is totally ordered (with ordering a_1, \dots, a_n) by $\Delta(\sigma)$
3. a_n precedes all actions in $A(\sigma) \setminus \{a_1 \dots a_n\}$

Then $\rho_{pre}(\sigma)$ is the plan prefix of σ , and $s_{pre}(\sigma) = (\rho_{pre}(\sigma))(I)$ is the prefix state of σ .

If the prefix state $s_{pre}(\sigma(P))$ of an approach $\sigma(P)$ is the goal state, then that approach is a plan solving P . For a forward state space planner, all of the actions in any approach considered will be in the plan prefix of the approach.

Next, that portion of an approach corresponding to a plan suffix must also be identified, to establish a state for backward propagation. Unfortunately, this definition is a bit trickier, requiring

some new machinery. In order to say what state corresponds to the beginning of the plan suffix, the notion of regression from the goal state, touched upon in chapter 2 (p. 12), needs to be made explicit and formalized.

Definition 4.7 (Regression) *Given a plan ρ and a state $s_G \neq \perp$, the regression $\text{regress}(\rho, s_G)$ of s_G through ρ is any minimal state s_1 under the subset partial ordering such that $\rho(s_1) = s_G$, or \perp if no such state exists.*

The regression $\text{regress}(\rho, s_G)$ is always uniquely defined. To see this, it is actually sufficient to consider the reversal technique described in chapter 3; the state $\rho_R(s_G^s)$ is unique, and omitting all DC effects from it leads to the desired regression. The details are complicated, however, so a more direct proof is given instead.

Lemma 4.1 (Regression Uniquely Defined) *For any plan ρ and state $s_G \neq \perp$, $\text{regress}(\rho, s_G)$ is unique.*

PROOF:

By definition, when $r = \text{regress}(\rho, s_G) = \perp$, it is unique. Suppose that $r \neq \perp$. If there is a unique state s_1 such that $\rho(s_1) = s_G$, then it is the unique minimal state.

Otherwise, suppose that there are two states s_1 and s_2 , both minimal, such that $s_1 \neq s_2$ and $\rho(s_1) = \rho(s_2) = s_G$. Both states must supply the necessary initial conditions for ρ to produce s_G . Thus their intersection $s_{12} = s_1 \cap s_2$ must also provide the necessary initial conditions for ρ to produce s_1 . But the intersection of two sets must be a subset of both, so $s_{12} \subseteq s_1$ and $s_{12} \subseteq s_2$. Since $s_1 \neq s_2$ at least one of s_1 and s_2 is a proper superset of s_{12} , so s_1 and s_2 cannot both be minimal, contradicting the earlier assumption.

Given an identification of that portion of an approach that is a linear plan and whose actions occur at the end of the approach, having a well-defined definition of regression in hand allows identification of a state from which backward propagation can proceed.

Definition 4.8 (Plan Suffix) *Given an approach σ to a planning problem with goal G , let $\rho_{suf}(\sigma) = \langle a_1, \dots, a_n \rangle$ be the longest sequence of actions such that*

1. $a_1 \dots a_n \in A(\sigma)$
2. $\rho_{suf}(\sigma)$ is totally ordered (with ordering a_1, \dots, a_n) by $\Delta(\sigma)$
3. a_1 succeeds all actions in $A(\sigma) \setminus \{a_1 \dots a_n\}$

Then $\rho_{suf}(\sigma)$ is the plan suffix of σ , and

$$s_{suf}(\sigma) = \text{regress}(\rho_{suf}(\sigma), G)$$

(definition 4.7) is the suffix state of σ .

Given these concepts, a workable form of propagation can be described. The idea is that, if a given action is the only way to proceed forward from a prefix state of an approach, or backward from a suffix state, then any successful development of this approach to a plan will contain this action at this point: it may as well be added directly. Dealing with prefix states or suffix states avoids having to make complex logical inferences to decide what propagation is allowed.

Forward propagation, therefore, will be defined as adding the sole action to an approach that is legal in the prefix state.

Definition 4.9 (Forward Propagation Step) Given an approach σ , let $A_{pre}(\sigma)$ be the set of actions legal in $s_{pre}(\sigma)$:

$$A_{pre}(\sigma(P)) = \{a \in A(P) \mid a(s_{pre}(\sigma(P))) \neq \perp\}$$

When $A_{pre}(\sigma) = \emptyset$, the approach has failed. When $A_{pre}(\sigma) = \{a\}$, a forward propagation step can be taken in σ . A forward propagation step is a development σ' of σ . The action set $A(\sigma')$ is $A(\sigma) \cup A_{pre}(\sigma)$, and the partial order $\Delta(\sigma')$ is the union of $\Delta(\sigma)$ with ordering constraints that append a to the actions of $\rho_{pre}(\sigma)$, and make it precede the other actions in σ' .

Several issues arise here. The definition is constructed so that forward propagation steps can be usefully iterated: a forward propagated action becomes part of the plan prefix, permitting new propagations. Forward propagation does not affect the soundness of search space planning: the action and ordering constraints added to an approach by a forward propagation step will not affect the legality of plans resulting from the approach. Forward propagation also does not affect the completeness of search space planning: since the forward propagated actions are the only legal actions that can occur at the prefix state of the approach, they can be added to the approach at the prefix state without changing what legal plans will ultimately be produced from it.

A backward propagation step is similar to the forward one: it adds an action to the head of the plan suffix.

Definition 4.10 (Backward Propagation Step) Given an approach σ , let $A_{suf}(\sigma)$ be the set of actions that $s_{suf}(\sigma)$ can be regressed through:

$$A_{suf}(\sigma(P)) = \{a \in A(P) \mid \text{regress}(a, s_{suf}(\sigma(P))) \neq \perp\}$$

When $A_{suf}(\sigma) = \emptyset$, the approach has failed. When $A_{suf}(\sigma) = \{a\}$, a backward propagation step can be taken in σ . A backward propagation step is a development σ' of σ . The action set $A(\sigma')$ is $A(\sigma) \cup A_{suf}(\sigma)$, and the partial order $\Delta(\sigma')$ is the union of $\Delta(\sigma)$ with ordering constraints that prepend a to the actions of $\rho_{suf}(\sigma)$, and make it succeed the other actions in σ' .

As with forward propagation, backward propagation steps are iterable, sound, and complete, and for the same reasons.

4.1.3 Propagating Planners

Having given definitions of forward and backward propagation, it is time to return to the original question: How can the restrictions of strong forward and backward planning be weakened to allow propagation? The answer is now evident: simply allow opposite-directional propagation steps.

Definition 4.11 (Forward Planner) A forward planner is a search space planner in which a development of an approach is constructed only if it is a forward search step (definition 4.4) or a backward propagation step (definition 4.10).

Definition 4.12 (Backward Planner) A backward planner is a search space planner in which a development of an approach is constructed only if it is a backward search step (definition 4.5) or a forward propagation step (definition 4.9).

This addresses the objection that the definitions may be too restrictive: general forward and backward planners will be treated in the remainder of this chapter.

4.2 One-Way Functions

A detailed treatment of cryptography is outside the scope of this work. For a fuller view of its principles and techniques, consult an introductory text such as Schneier’s [51] or Stinson’s [55]. This work is concerned only with a certain class of cryptographic algorithms, for which the following definition captures the essential properties:

Definition 4.13 (One-Way Function) *A cryptographic hash function or one-way function H is a binary function from m input bits to n output bits, with the property that it is tractable to compute but intractable to invert. Specifically, computing an output given an input ($y = H(x)$) is in P , while computing an input given an output ($x = H^{-1}(y)$) is not in P .*

Under some fairly simple assumptions, one can prove that one-way functions must exist if (and only if) $P \neq NP$ [4].²

It is believed by many researchers that computing inverses for one-way functions in common use in cryptography, such as MD5 [51], is NP-hard. In any case, the ability to efficiently compute these inverses would have such important practical consequences that the following assumption may be safely employed.

Assumption 4.1 (One-Way Function Inversion) *A family of one-way functions $H : 2^n \rightarrow 2^n$ is constructible, and has the property that H can be computed in time $O(n)$, but H^{-1} can be computed only in time $\omega(2^n)$.*

4.3 Boolean Circuits

The notion of a Boolean circuit is crucial to expressing one-way functions in a form useful for planning.

Definition 4.14 (Boolean Circuit) *A Boolean circuit is “a directed acyclic graph (DAG) whose vertices are labeled with the names of Boolean functions (logic gates) or variables (inputs [and outputs])” [50]. For our purposes, a Boolean circuit is a DAG $C = \langle V, E \rangle$. The vertices V are gates and are labeled according to their function, as defined below. The edges E are lines, each of which connects an output to an input.*

The function of a logic gate is defined by a map, or truth table, giving for each legal combination of Boolean inputs of the gate (value of in-edges of the vertex), the Boolean outputs of the gate (values of out-edges of the vertex). The edges of a Boolean circuit graph are called lines and are labeled with signals.

The truth table of a logic gate v is a map T_v from each element of the set of conjunctive formulae (definition 1.3) over the input signals of v to a conjunctive formula over the output signals of v .

For example, the truth table of a Boolean AND gate v with input signals i_0 and i_1 and output q is

$$T_v = \begin{cases} \{\neg q\} & \text{inputs}(v) = \{\neg i_0, \neg i_1\} \\ \{\neg q\} & \text{inputs}(v) = \{\neg i_0, i_1\} \\ \{\neg q\} & \text{inputs}(v) = \{i_0, \neg i_1\} \\ \{q\} & \text{inputs}(v) = \{i_0, i_1\} \end{cases}$$

²Note that the definition given here does not require a one-way function to be one-to-one. Again, see Balcázar, Díaz, and Gabarró [4] for details.

For any class of functions F in P , there is a uniform³ class of Boolean circuits that compute the functions in F . Of particular interest is that for any one-way function H there is a one-way Boolean circuit that computes H .

In this work, the definition of logic gate will actually be extended a bit. Rather than limit a gate to have a given output for a given input, the extended definition will allow a gate to nondeterministically select from a number of outputs for a given input. This modification turns out to be convenient for the purposes of completing the analogy between a gate and a set of actions in a plan.

Definition 4.15 (Nondeterministic Boolean Circuit) A Nondeterministic Boolean Circuit (NBC) is a Boolean circuit, except that the truth table T_v of an NBC gate v is a map from each possible input to a set of possible outputs. Evaluation of a gate involves nondeterministically selecting an element of that set as the output of the gate.

So, for example, an NBC gate v with input i and output q might have the truth table

$$T_v = \begin{cases} \emptyset & \text{inputs}(v) = \{\neg i\} \\ \{\{q\} \quad \{\neg q\}\} & \text{inputs}(v) = \{i\} \end{cases}$$

Such a gate can evaluate successfully only if its input is true, in which case its output may be either true or false.

The advantage of working with NBCs is that there is a straightforward correspondence between NBCs and planning problems. This correspondence will be heavily exploited in understanding the constructions of this chapter.

4.3.1 NBC Planning Problems

NBC Planning Problems (NBCPPs) are a particular class of planning problems associated with a given NBC, such that evaluating the NBC and solving a planning problem in the class are isomorphic tasks. The basic idea behind the NBCPP construction is to use a set of actions for each gate that represent its possible evaluations. Exactly one action from this set must be chosen, at the appropriate point, in building a legal plan for the problem: the action chosen corresponds to the evaluation of the gate. Thus, the entire plan consists of a topologically ordered sequence of gate evaluations.

Definition 4.16 (NBCPP) For any nondeterministic Boolean circuit $C = \langle V, E \rangle$, with the gates labeled in a given topological sort order $V = \langle v_1, \dots, v_n \rangle$, the nondeterministic Boolean circuit planning problem $\langle \langle F, A \rangle, I, G \rangle$ is constructed as follows.

Let $\text{in}(C)$ be the set of inputs of C . Let $\text{out}(C)$ be the set of outputs of C . Define the fluents of the problem by

$$F = (\text{in}(C) \cup \text{out}(C))^{\pm} \cup \{do_1 \dots do_{n+1}\}$$

(Recall that definition 3.1 gives the meaning of S^{\pm} .) Define the initial state of the problem by $I = \text{in}^s(C) \cup \{do_1\}$, and the goal state of the problem by $G = \text{out}^s(C) \cup \{do_{n+1}\}$.

³Intuitively, uniformity captures the notion that a circuit of a particular size is easily describable. For a more detailed explanation, see Savage [50]. All circuit classes used in this chapter are uniform.

Actions of the problem will be defined by the truth table of its gates, as follows. Given conjunctive formulae α and β , let the action $a_{i\alpha\beta}$ that requires α and produces β in vertex v_i be defined by

$$a_{i\alpha\beta} = \frac{\alpha^s \text{ do}_i}{\beta^s \neg\text{do}_i \text{ do}_{i+1}}$$

Then given a gate v_i with truth table T_{v_i} , and an input $\alpha \in \text{dom}(T_{v_i})$, the set of actions $A_\alpha(v_i)$ associated with input α is

$$A_\alpha(v_i) = \{a_{i\alpha\beta} \mid \beta \in T_{v_i}(\alpha)\}$$

The set of actions A for the entire domain is the set of actions for all inputs of all gates.

$$A = \bigcup_{v_i \in V, \alpha \in \text{dom}(T_{v_i})} A_\alpha(v_i)$$

The key fact is that solving a NBCPP corresponds to evaluating the corresponding NBC.

Lemma 4.2 (NBC Evaluation Planning) *Given a NBC C and a corresponding NBCPP $P(C)$, a plan ρ (with $|\rho| = |V(C)|$) solving $P(C)$ exists if and only if C computes the given outputs from the given inputs.*

PROOF: *By dual implication.*

Circuit solution \implies plan existence:

Consider the input values α and output values β at any gate $v_i \in C$, and the corresponding set of actions $A_\alpha(v_i)$ (definition 4.16). By construction, there must be an action $a_{i\alpha\beta} \in A_\alpha(v_i)$ of the form

$$\frac{\alpha^s \text{ do}_i}{\beta^s \neg\text{do}_i \text{ do}_{i+1}}$$

Call this action $a(v_i)$. Thus, a plan can be constructed as follows: Begin with the empty plan ρ_0 . Now, for each vertex v_i of C in turn, append action $a(v_i)$ to ρ_{i-1} to obtain ρ_i . Since the vertices occur in topological sort order, each of the inputs of each vertex v_i must be drawn only from the outputs of earlier vertices v_1, v_2, \dots, v_{i-1} , and therefore the input fluent values α of $a_{i\alpha\beta}$ are fixed. Further, the precondition do_i will be satisfied, either by the effect of the previous action for v_{i-1} or by the initial state. Thus, the action with output β can be chosen. The plan will end with an action for v_n , at which time the goal conditions will be satisfied, so ρ is valid, and $|\rho| = |V|$.

Plan existence \implies circuit solution:

Consider the preconditions and effects of any action $a_{i\alpha\beta}$ in ρ . By definition, a corresponding gate v_i is in C , and given input α can produce output β . Thus, the circuit can be evaluated as follows: The last action $a_n = a_{n\alpha\beta}$ of ρ must produce only goal conditions, and must correspond to the appropriate evaluation of gate v_n of C . Set the inputs and outputs of v_n according to α and β . Now, consider each action $a_{i\alpha\beta}$ producing an input precondition of a_n . Each must correspond to a gate v_i whose output is an input of v_n . Set the inputs of v_i to α , and its outputs to β . Continue until all values of all gates of C have been set. This must happen, since the fluents of P have been uniquely labeled according to the signals of C . Thus, the evaluation of C is correct.

For deterministic Boolean circuits, the plan for a given NBCPP will be unique. For nondeterministic ones, there may be multiple possible evaluations of the circuit: each will correspond to a unique plan.

4.3.2 NBCPPs and Reversal

It is interesting and useful to consider how the reversal of planning problems discussed in the previous chapter interacts with the NBCPP construction. To do this it is helpful to consider a construction whose plans are the reversal of circuit evaluations. Note that the reversal technique of chapter 3 could be used to obtain such a construction from the forward version. Unfortunately, the actions of definition 4.16, when reversed, contain DC effects that must be compiled out using C_R . While this is possible, it obscures the symmetry of the forward and backward problems, and leads to larger (and thus more difficult) problems. The construction given here avoids this, creating backward planning problems which are similar to the forward problems of definition 4.16.

Definition 4.17 (Reverse NBCPP) *The reverse NBCPP associated with a circuit C is constructed by reversing the initial and goal states of the NBCPP for C , and the input and output fluents of the actions. Given definition 4.16, define instead the initial state of the problem by $I = \text{out}^s(C) \cup \{\text{do}_{n+1}\}$, and the goal state of the problem by $G = \text{in}^s(C) \cup \{\text{do}_1\}$.*

Given conjunctive formulae α and β , let the action $\hat{a}_{i\alpha\beta}$ be defined by

$$\hat{a}_{i\alpha\beta} = \frac{\beta^s \text{do}_{i+1}}{\alpha^s \neg \text{do}_{i+1} \text{do}_i}$$

Then given a gate v_i with truth table T_{v_i} , and an input $\alpha \in \text{dom}(T_{v_i})$, the set of actions $\hat{A}_\alpha(v_i)$ associated with input α is

$$\hat{A}_\alpha(v_i) = \{\hat{a}_{i\alpha\beta} \mid \beta \in T_{v_i}(\alpha)\}$$

Then the set of actions A for the entire domain is the set of actions for all inputs of all gates.

$$A = \bigcup_{v_i \in V, \alpha \in \text{dom}(T_{v_i})} \hat{A}_\alpha(v_i)$$

The important thing about definition 4.17 is that the reversal of a plan for $P_R(C)$ is isomorphic to an evaluation of C .

Lemma 4.3 (Reverse NBC Evaluation Planning) *Given an NBC C and a corresponding reverse NBCPP $P_R(C)$, a plan ρ_R (with $|\rho_R| = |C|$) solving $P_R(C)$ exists if and only if C computes the given outputs from the given inputs.*

PROOF: *By dual implication.*

Circuit solution \implies plan existence:

Consider the input values α and output values β at any gate $v_i \in C$, and the corresponding set of actions \hat{A}_α (definition 4.17). By construction, there must be an action $\hat{a}_{i\alpha\beta} \in \hat{A}_\alpha(v_i)$ (and therefore in \hat{A}) of the form

$$\frac{\beta^s \text{do}_{i+1}}{\alpha^s \neg \text{do}_{i+1} \text{do}_i}$$

Thus, a plan can be constructed as follows: Begin with the empty plan ρ_0 . Now, for each vertex v_i of C in turn, prepend action $\hat{a}_\alpha(v_i)$ to ρ_{i-1} to obtain ρ_i . Since each of the inputs of each vertex v_i must be drawn only from earlier vertices v_1, v_2, \dots, v_{i-1} , an action \hat{b} that satisfies the precondition β^s of $\hat{a}_{i\alpha\beta}$ as required will occur earlier in the final plan, and since fluents are uniquely labeled by signals, no action between \hat{b} and \hat{a} will delete this precondition. The do_i conditions will be satisfied as well, by the way that the actions are ordered. Thus ρ_R is valid and $|\rho_R| = |V|$.

Plan existence \implies circuit solution:

Consider the preconditions and effects of any action $\hat{a}_{i\alpha\beta}$ in ρ_R . By definition, a corresponding gate v_i is in C , and given input α produces output β . Thus, the circuit can be evaluated as follows: The first action $a_1 = \hat{a}_{n\alpha\beta}$ of ρ must require a fluent of the initial state, and must correspond to an output gate v_n of C . Set the inputs and outputs of v_n according to α and β . Now, consider each action a consuming an effect of a_1 . Each must correspond to a gate v_i whose output is an input of v_n . Set the inputs of each v_i according to the corresponding a . Continue until all values of all gates of C have been set. This must happen, since the fluents of P have been uniquely labeled according to the signals of C . Thus, the evaluation of C is correct.

4.4 A Planner Directionality Test

Given all of the machinery introduced above, classes of forward and backward PROPS planning problems can be constructed. These classes will have the property that a forward planner (according to the definitions given above) will find the forward problems easy but the backward problems intractable. Similarly, a backward planner will find the backward problems easy and the forward problems intractable. After describing the construction of these directional problems, a proof of the correctness of the construction is outlined. Finally, the features and limitations of the implementation used in this work are discussed.

4.5 Construction

The basic idea of the construction is to take a one-way Boolean circuit and transform it into a planning problem using the construction of definition 4.16. A planner that tries to start from the wrong end of the planning problem will thus be attempting to compute an input to the one-way function from an output: by assumption 4.1, this is intractable. A planner that works from the correct end, on the other hand, will be computing an output of the one-way function from an input: this should be easy.

The Boolean circuit, and thus the planning problem, can be oriented so that the input to the one-way function corresponds to an unspecified initial state, and the output corresponds to an unspecified goal state, producing a *forward problem*. Alternatively, in a *backward problem* the input to the one-way function corresponds to an unspecified goal state, and the output to an unspecified initial state. (Unspecified initial states, as well as unspecified goal states, can be produced from total states in a simple fashion, as exhibited below.) Figure 4.1 illustrates this concept: the arrow indicates the easy direction for the one-way function.

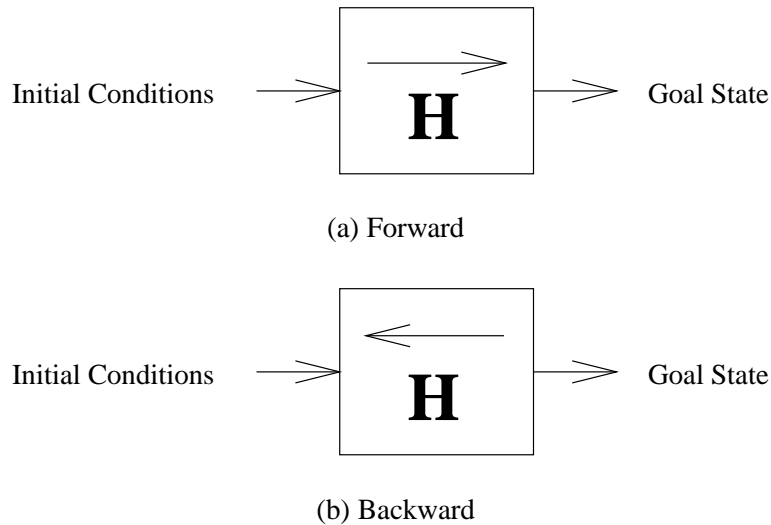


Figure 4.1: One-Way Function Planning Problems

4.5.1 Bit Commitment

A subtle problem arises at this point. Consider the behavior of a backward search space planner searching depth-first to produce a plan for a forward one-way function planning problem. By the argument above, this planner should find this problem infeasible. The planner, however, may proceed as follows:

1. Select an open goal bit, and a value for that bit.
2. Chain backward through the problem using any matching operator, until selecting an action that can be first in the resulting plan. The planner can easily check this condition, by verifying that all the preconditions of the chosen operator are satisfied in the initial state.
3. If chaining cannot proceed, discard the actions selected for this goal bit, and start at step 2 with the other value for the open goal bit.
4. Include the chosen action as the first action in a partial plan.
5. Forward-propagate the chosen action to completion.
6. Repeat steps 1—5 until all goal bits are chosen.

Since the depth of the planning problem is polynomial, and the size of the resulting plan is polynomial, this is a polynomial-time algorithm for solving this problem.

This catch arises because it is emphatically *not* an intractable problem to find an input to a one-way function that produces a given output *bit*, and because the definition of backward planner allows such a planner to work its way to the front starting from this bit. The offhand solution is to modify the definition of backward planner: this is unacceptable, however, since many backward

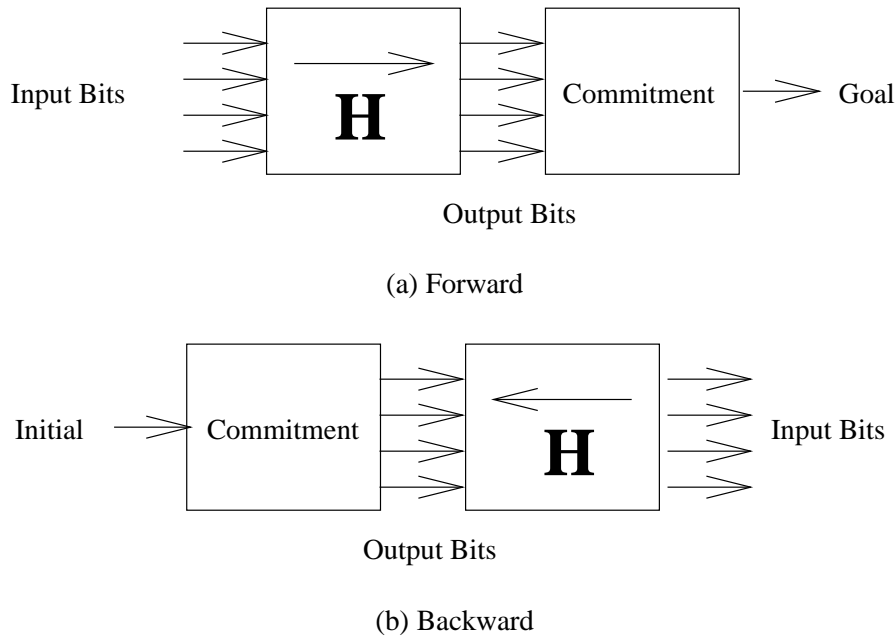


Figure 4.2: One-Way Planning Problem with Bit Commitment

planners actually search in this fashion—this behavior was discovered by examination of execution traces of UCPOP.

In order to take advantage of one-way problems to detect planner directionality, it will apparently be necessary to force a wrong-directional planner to commit to *all* of the output bits of the one-way function before proceeding any further. Figure 4.2 illustrates this concept.

A construction that explicitly achieves this commitment property can be seen to be impossible. Consider the operator or operators that actually assign values to the output fluents of the one-way function. If some operator sets only some of the fluents, a planner can proceed as above by working its way to that operator. If each operator sets all of the fluent values simultaneously, either there is an exponential number of operators or the problem is intractable for any planner, since it requires predicting the input that will produce a given output from the one-way function.

All is not lost, however: it is possible to construct the problem in such a way that any search tree that commits to an output bit has *implicitly* committed to all output bits: that is, there is no extension of an approach that sets an output bit to a plan in which the other output bits take on any but a single value.

4.5.2 Bit Commitment Network

The *bit commitment network* used to correct the above problem is probably best described in terms of the correspondence between the circuit and PROPS representations. Only the forward version (i.e., the version that follows a forward one-way function and forces a backward planner to commit) will be described here: the backward version is simply the reversal of the forward one.

Table 4.1: Truth Table for Left Projection Gate

L	R	Q
<i>f</i>	<i>f</i>	<i>f</i>
<i>f</i>	<i>t</i>	<i>f</i>
<i>t</i>	<i>f</i>	<i>t</i>
<i>t</i>	<i>t</i>	<i>t</i>

$$\frac{L \ R}{Q} \quad \frac{L \ \neg R}{Q} \quad \frac{\neg L \ R}{\neg Q} \quad \frac{\neg L \ \neg R}{\neg Q}$$

Figure 4.3: Operators for Left Projection Gate

The fundamental gate and corresponding set of operators used in this network is the *left projection* gate, whose output is the same as its left input—the right input is ignored. The truth table for a left projection gate with inputs L and R and output Q is given by table 4.1, and leads by the construction of definition 4.16 to the operators of figure 4.3. Graphically, in what follows, the operators will be represented as in figure 4.4.

The basic idea behind the construction is that a backward search space planner, in selecting a particular left projection gate operator to match a given output Q , must commit to a particular value of both L and R . Thus, given a commitment to a single fluent value, a commitment to two fluent values can be obtained. Layers of left projection gates can be assembled to continue this effect: after n layers, commitment to $n + 1$ bits will be achieved. Each left projection gate by is labeled by the bit it commits to and its layer within the network. The fluents are named by the gate producing them. Figure 4.5 shows the labeling of gates, inputs, and outputs, in the form ‘*bit/layer of graph*’. The structure of figure 4.6 uses this labeling, and provides a commitment to 4 bits. The fluents on the left hand side of figure 4.6 will be referred to as the *inputs* to the bit commitment network, and the single fluent on the right hand side as the *output* from the network. (Thus, for a reversed network, the inputs would be on the right and the output on the left.) This convention is arbitrary, but is motivated by the notion that the outputs of the one-way function are the inputs to the commitment

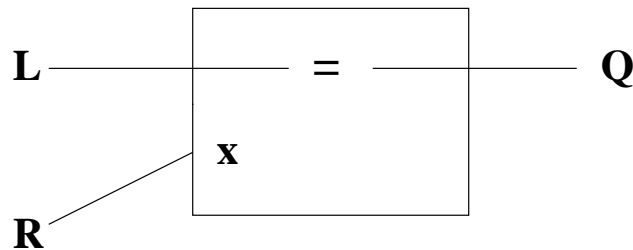


Figure 4.4: Schematic for Left Projection Gate



Figure 4.5: Gate and Fluent Labeling by Bit/Layer

network.

The reverse bit commitment network planning problem is constructed in exactly the same fashion as the forward one, but using the reverse NBCPP construction of lemma 4.3 that produces bit commitment when planning in the opposite direction.

4.5.3 Final Construction

The final construction of the forward tractable search problems proceeds by concatenating three components: a circuit for producing a nondeterministic value, a circuit for a one-way function, and the bit-commitment network of the previous section. The result is a one-way Boolean circuit that can be translated to a one-way planning problem via definition 4.16.

The nondeterministic value circuit consists of a gate that, for each input, may produce either a true or a false value. Note that this is the truth table of a previous example (p. 40).

$$T_v = \begin{cases} \emptyset & \text{inputs}(v) = \{\neg i\} \\ \{\{q\} \quad \{\neg q\}\} & \text{inputs}(v) = \{i\} \end{cases}$$

This is the motivation for using NBCs instead of Boolean circuits: this sort of gate can be naturally represented as a set of actions within the planning framework.

The forward tractable one-way planning problem construction is given by figure 4.7. A forward search space planner may select values for the inputs to the one-way problem by picking either action for the nondeterministic value gates, forward propagating these inputs through the one-way function, and finally forward propagating the one-way outputs through the bit commitment network. A backward search space planner, on the other hand, in working backward through the bit commitment network, will implicitly commit to a particular output value of the one-way function, and thus commit itself to solving an intractable problem. (Note that the planner cannot forward-propagate: this would be allowed by definition 4.9 except that the nondeterministic value gates inhibit this propagation.)

The reverse tractable problem is exactly the forward tractable problem, except constructed using the reverse NBCPP construction of definition 4.17. Figure 4.8 shows the conceptual design of this problem. Note that this figure illustrates the nondeterministic consequences of the reversal: the nondeterministic value gates will translate into deterministic actions, whereas the bit commitment gates will now be nondeterministic in their outputs.

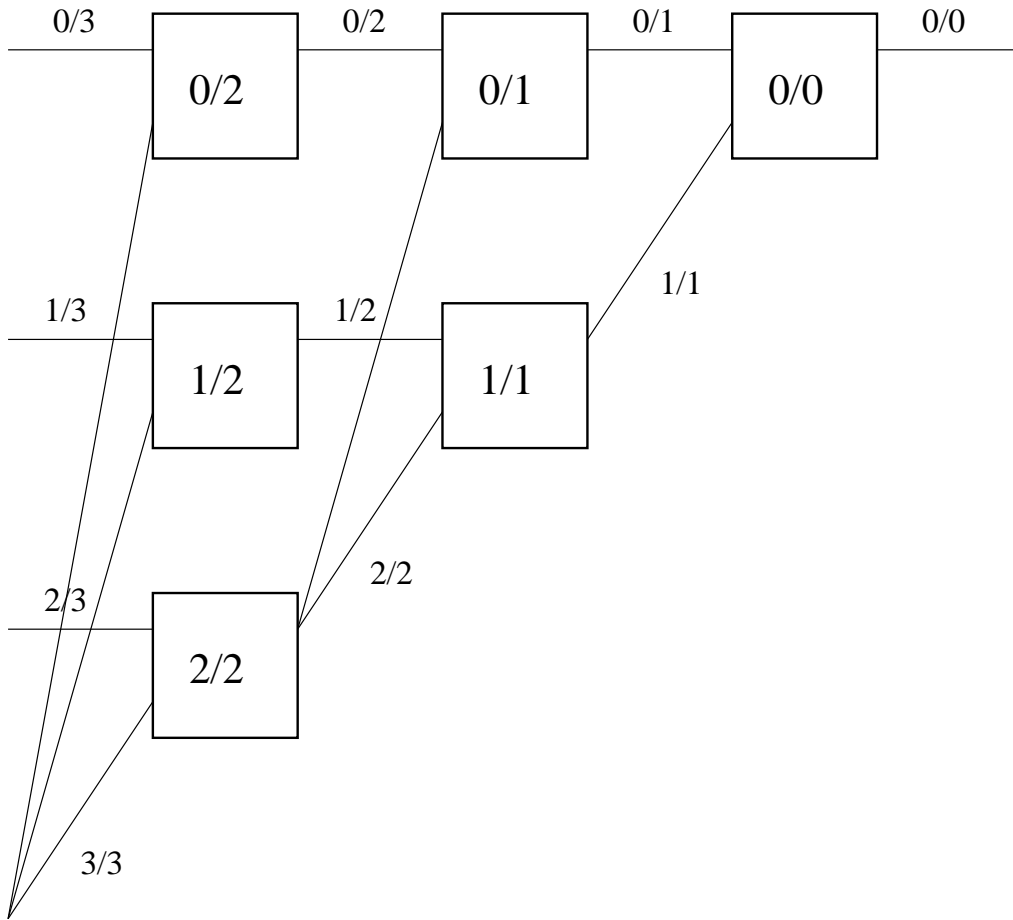


Figure 4.6: 4-Bit Commitment Network

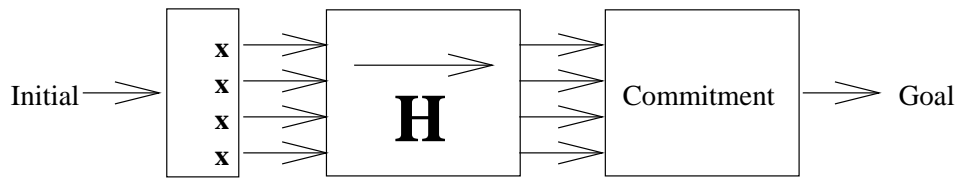


Figure 4.7: Forward One-Way Planning Problem

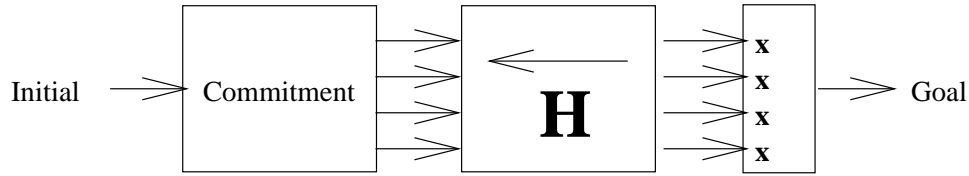


Figure 4.8: Backward One-Way Planning Problem

4.6 Correctness

To see that the planning problems as constructed will correctly detect the direction of a sufficiently powerful search space planner, it is sufficient to show that

1. Given a commitment to its input bits, the one-way function portion of the problem is tractable in the easy direction.
2. Given a commitment to its output bits, the one-way function portion of the problem is intractable in the hard direction.
3. The bit commitment network portion of the problem will commit to the output bits of the one-way function.
4. The bit commitment network is tractable in the easy direction.

It is difficult to prove (1) for arbitrary search space planners, since deliberately bad planners are capable of getting lost on any problem where there are choices. However, blindly fixing the inputs to the one-way portion of the problem and then merely propagating repeatedly will yield the outputs. It seems that any reasonable planner should be able to at least emulate this strategy.

Lemma 4.4 (One-Way Evaluation Easy) *A search space planner working in the easy direction and not committed to a given output can discover a plan for a n bit one-way planning problem P in time asymptotically less than $\mathcal{O}(n^k)$.*

PROOF: *By construction.*

Consider the situation after a planner working in the easy direction has chosen an arbitrary input to the one-way circuit. Since the circuit is deterministic, this means that the operators to choose for the next layer of gates in the one-way circuit are uniquely determined. But by the definition of directional propagation (definitions 4.9 and 4.10) this means that a propagating planner can proceed to the output with no backtracking. Since the problem is polysize, a non-backtracking solution will be obtained in polytime.

To prove (2), it is sufficient to appeal to assumption 4.1.

Lemma 4.5 (One-Way Inversion Hard) *No search space planner committed to a given output can discover a plan for a n bit one-way planning problem P in time asymptotically less than $\mathcal{O}(2^n)$.*

PROOF: *By contradiction.*

Given a commitment to a given set of n output bits, assume that a search space planner can find a plan ρ for P in time asymptotically less than $\mathcal{O}(n^k)$. By the construction of P and lemma 4.2 or 4.3, the input bits for the one-way circuit C generating P can be read from ρ in time $\mathcal{O}(n)$. Thus, the input to the one-way function H generating C such that H on this input produces the given output has been determined in polynomial time. But this contradicts assumption 4.1.

To prove (3), it is sufficient to show that the bit commitment network works in the fashion described informally previously (pp. 45–47).

Lemma 4.6 (Bit Commitment Network Commits) *Consider a search space planner (definition 4.2) P working from the output to the inputs of a n -bit commitment network. If P is currently considering an approach σ that contains an operator that fixes the value of some input bit $i = B_i$, then there exists $B_0 \dots B_{n-1}$ such that any development of σ into a legal plan will have*

$$\forall i \in \{0, \dots, n-1\} . \text{input bit } i = B_i$$

PROOF: *By induction on the number n of bits of the bit commitment network.*

Base case:

When $n = 1$, there is only one input bit, and no network. This sole bit is clearly fixed by the planner.

When $n = 2$, there appear to be two cases: the planner has fixed input bit 0, or the planner has fixed input bit 1. However, in either case the planner must have done so by selecting an operator to implement the sole left projection gate in the network: this selection must have committed to both input bits.

Inductive case:

Assume that the lemma is true for an n bit commitment network. Then, upon reaching any input $0/n - 1 \dots n - 1/n - 1$ to layer $n - 1$, the planner will have committed to all of these inputs. A search space planner searching from output to inputs must, by definitions 4.12 and 4.11, select operators for layer n based on using the output of each gate i/n to satisfy the input of layer $n - 1$. By the inductive hypothesis, this means that layer n is committed to exactly the same values for bits $0..n - 1$ as layer $n - 1$. Further, once some operator for gate i/n has been selected in layer n , the value of fluent n/n is committed to by this selection: any legal plan developed from an approach containing this operator must have selected operators for the other gates in layer n consistent with the value of fluent n/n . Thus, upon reaching any input $0/n \dots n/n$ to layer n , the planner will have committed to all of these inputs, implying that the lemma is true for a $n + 1$ bit commitment network.

Finally, as with (1), (4) is difficult to prove for arbitrarily bad planners. Essentially the same sort of propagation argument as lemma 4.4 holds, however: given a fixed input, simple directional propagation (definitions 4.12 and 4.11) suffices to plan through the network.

4.7 Conclusions

The beginning of this chapter posed three questions. Two of them have been answered. A reasonable formal account of the behavior a large class of planning algorithms has been given using the

notion of a search space planner (definition 4.2), and a reasonable definition of directionality for search space planners has been provided (definitions 4.11 and 4.12). Directional classes of planning problems have been described, such that the performance of a directional planner of sufficient power will necessarily scale differently on the problems in the forward and backward classes. A proof has been sketched of the correctness of this result.

To answer the third question, a variant of the directional problem classes described here has been implemented, and the performance of a variety of existing planners has been measured. Chapter 5 describes the implementation, and the results of these experiments.

Chapter 5

Measuring Planner Direction

In chapter 4, a construction was given for classes of forward and backward one-way planning problems. A forward one-way planning problem is easy for a forward search space planner and intractable for a backward search space planner; a backward one-way planning problem is easy for a backward search space planner and intractable for a forward search space planner.

In this chapter, a specific class of directional one-way planning problems is described, based upon a novel hash function. Next, the directionality of a variety of planners is evaluated using these problems, both to verify the technique, and to better understand the behavior of the planners themselves.

5.1 Implementation

The construction given in chapter 4 (p. 47) for one-way planning problems is quite specific, excepting its failure to prescribe a particular one-way function to use as the core of the problem. Ideally, a one-way function believed to be cryptographically sound, such as MD5, should be used in determining planner directionality. A cryptographic quality one-way function is required by the proofs in chapter 4 that depend on assumption 4.1.

Unfortunately, cryptographically sound one-way functions tend to have large circuits. The corresponding planning problem would require tens or hundreds of thousands of operators, far more than current planners can deal with efficiently. Further, these cryptographically sound one-way functions tend to have deep circuits: this would lead to corresponding planning problems requiring long solutions, and experience indicates that such deep planning problems are especially difficult for existing planners. Finally, and most importantly, the only empirical way to estimate the asymptotic running time of the planner, and thus determine the direction, is to give the planner a class of problems of increasing size. Most existing cryptographic hash functions are of fixed size.

For all these reasons, a new scalable cryptographic hash function of modest size and fixed shallow depth was designed for experimentation with current planners.

Definition 5.1 (Hash Function \mathcal{H}_n) Let \mathbf{x} and \mathbf{y} be n -bit vectors, with n even. Let \mathbf{x}_H denote the high-order $n/2$ bits of \mathbf{x} , \mathbf{x}_L denote the low-order $n/2$ bits, and similarly for \mathbf{y}_H and \mathbf{y}_L . Finally, let A and B each be a random function (or lookup table) from $n/2$ bits to $n/2$ bits. The function

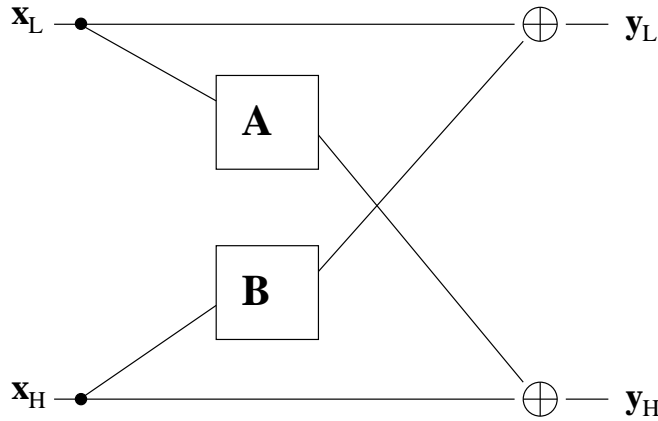


Figure 5.1: One-Way Function \mathcal{H}_n

$\mathcal{H}_n(\mathbf{x}) = \mathbf{y}$ is defined by

$$\begin{aligned} \mathbf{y}_H &= \mathbf{x}_H \oplus A(\mathbf{x}_L) \\ \mathbf{y}_L &= \mathbf{x}_L \oplus B(\mathbf{x}_H) \end{aligned}$$

where \oplus is the exclusive-or operation (addition in $GF(2)$).

Figure 5.1 illustrates this function.

\mathcal{H}_n solves the problems discussed above. Expressed as a planning problem, it requires $\mathcal{O}(2^n)$ operators, but with a sufficiently small constant factor to be viable for reasonably large values of n . Its depth is constant with respect to n , which is extremely important. It is trivial to evaluate: the resulting planning problem, as shown below, is simple to solve in the easy direction.

Of course, all of this comes at a price. First, far from being intractable to invert, there is good reason to believe that a human mathematician could construct a closed-form \mathcal{H}_n^{-1} evaluable in polytime—certainly search techniques tuned to the problem can invert \mathcal{H}_n for a given output \mathbf{y} in approximately linear time. The function was constructed, however, using well-known techniques from elementary cryptography (diffusion and confusion); the authors' colleagues were unable to construct inverses offhand. Further, general-purpose planners are not good at this sort of thing: experimental tests with both the forward and backward problems show that the planners studied in this chapter do not find the inversion of \mathcal{H}_n tractable.

The second problem with \mathcal{H}_n is that the exponential growth in the number of operators makes the problem asymptotically intractable even in the easy direction. Again, experiments show that the planners studied in this section index operators well enough for this not to be too large a problem. However, future work should address the construction of better one-way functions; the existence of better planners would also solve the problem, by allowing the use of standard cryptographic hash functions.

At any rate, it is straightforward to transform \mathcal{H}_n into a planning problem. The functions A and B are each implemented by $2^{n/2}$ operators with a domain element as the preconditions and a random output as the effect. Only $4n$ operators are needed to implement the xor operations; 4 for the truth table for each xor, according to the construction of definition 4.16.

5.2 Experiments

Having derived a directional test for planners, and shown that it is workable in principle, all that remains is to apply it in practice: to actually feed the forward and backward version of the one way planning problem constructed in the previous section to planners. In this section, the directionality test will be applied to planners of the previous generation whose directionality is well understood, and also to recent planners that are not yet fully understood. In the process, some insight will be given into the performance of several modern planners.

5.2.1 Methodology

The hash function, commitment network, and associated machinery described previously have been implemented using a testbed composed mainly of M4 macros. M4 [30] is a powerful general purpose macro processor. The macro processor approach lends itself to implementation of the core primitives independent of the target syntax. A collection of M4 definitions is constructed for each new target planning language, describing that language's syntax for fluents, operators, and initial and goal states. A planner-independent collection of M4 support routines provides syntax for conveniently describing problems.

As noted previously (p. 42), both forward and backward versions of a problem are specified independently. This was deemed necessary to provide sufficient control over the implementation. The alternative, to automatically reverse a forward description, would require compiling the forward version to positive-only preconditions for comparability. This would make the problem descriptions larger, and their behavior more difficult to understand.

Implementation of the hash function \mathcal{H}_n requires the generation of the random functions A and B used in its specification (definition 5.1). The lack of a pseudo-random number generator in M4, plus the difficulty of expressing the sort of bit arithmetic necessary to generate and check the functions as M4 macros, made M4 an unlikely candidate for this portion of the task. Instead, random matrices are generated by a Java applet, that also selects (by enumeration of all possible inputs) an input/output pair $\langle x, y \rangle$ that is *one-to-one*: \mathcal{H}_n is deterministic and $\nexists x' \neq x . \mathcal{H}_n(x') = y$. In the process of selecting this pair, the Java applet collects statistics that show (although no formal statistical analysis has been performed) that \mathcal{H}_n remains reasonably collision-resistant as N grows: almost all outputs are produced by either 0, 1, or 2 inputs even for large N . The outputs of the applet consist of the collision statistics, a SAT formula version of the hash function used in testing its cryptographic strength, and a collection of M4 macros implementing the hash function for the testbed. This last is saved for reproducibility of results across experimental runs.

The structure of the testbed is illustrated in figure 5.2. In this diagram, solid boxes represent executable code: the source language is in parentheses. The dashed boxes represent data. The program/data distinction is somewhat arbitrary for a macro processor: in this diagram M4 is represented explicitly, and its inputs are shown as data. The dashed arrows represent invocation; downward dashed arrows with no source indicate human invocation. Solid arrows represent data flow. The testbed code is available for anonymous FTP [33]; it may well be useful in conducting other planning experiments, as well as providing a way to replicate and extend the results reported here.

All experiments were performed on a 450 MHz Intel Pentium II based machine with 256 MB of memory, under the Red Hat Linux operating system version 5.2 (Linux kernel version 2.0.36).

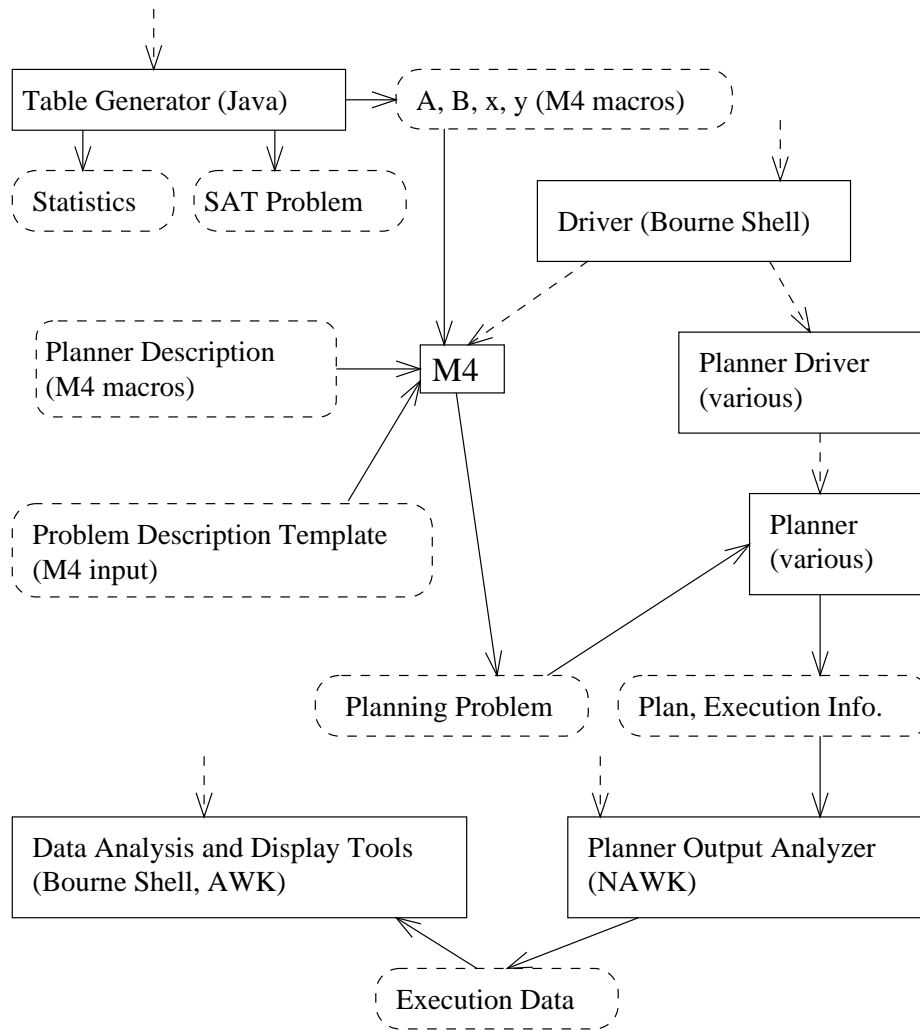


Figure 5.2: Detector Testbed Organization

Planners written in the C programming language were compiled with GCC version 2.7.2.3 at the highest optimization level, while planners written in the LISP programming language were compiled and executed using Allegro Common Lisp (ACL) Linux version 5.0, with speed and safety settings recommended by the planner authors. Times reported were CPU seconds in some cases and real-time seconds in others. Differences between these quantities appeared to be negligible in all experiments; as expected, the planners appeared to be completely CPU-bound.

The general philosophy behind the experiments was to attempt to obtain both a platform-independent measure of planner performance (some sort of node count) and a platform-dependent one (time). This was not feasible in all cases: in those cases where it was, there was good agreement between these measures, as seen below. The graphs in the next few sections show forward and backward planner performance on a log-linear scale, where the log scale is base 2. This is appropriate, since the number of operators in the planning problem roughly doubles for each increase of 2 bits in N . A linear graph would be expected for a successful planner under these conditions.

Five different experimental instances were generated for each bit size in each direction, and used uniformly across planners. None of these instances appears to be particularly anomalous, and good agreement was obtained between runs in all cases. This methodology is perhaps a bit questionable for $N = 2$: there are only 16 possible \mathcal{H}_2 functions in this case, and no effort was made to ensure the distinctness of the randomly chosen candidates. However, this has not proven to be a problem in practice.

5.2.2 Directionality of ASP

ASP [11] provides a nice introduction to the experimental results of this section, since its behavior is so straightforward. ASP plans using forward state space search guided by a goal distance heuristic. It is not complete: it may report failure on soluble problems if the heuristic is too difficult to calculate using available resources. In spite of its simple structure, ASP has been shown to be competitive with other modern planners such as `blackbox` and `Graphplan` on standard benchmark problems.

ASP proceeds in two stages: a planner generator stage generates a custom C search engine from the problem description, and this search engine is compiled and executed in a second stage to solve the planning problem.

Figure 5.3 shows the total time to solution (generation, compilation, and execution) for the forward problems. Note that ASP fails early on the $N = 10$ problems, due to its incompleteness. There is most likely a simple change to the resource allocation in the planning that would allow ASP to solve larger problems, but it was not evident after superficial study. Figure 5.4 shows just the execution time for the search engine, while figure 5.5 shows roughly the number of nodes searched in execution (actually the number of hash-table hits for A^* search).

Only forward times are shown: ASP is unable to solve the backward problem even for the $N = 2$ case! As a verification of the correctness of the problem description, all irrelevant actions in the backward $N = 2$ problem were deleted: ASP then found the expected plan in time comparable to the forward case. Thus the detector gives a clearcut determination of forward planning for ASP: this serves as a nice empirical validation of the experimental technique.

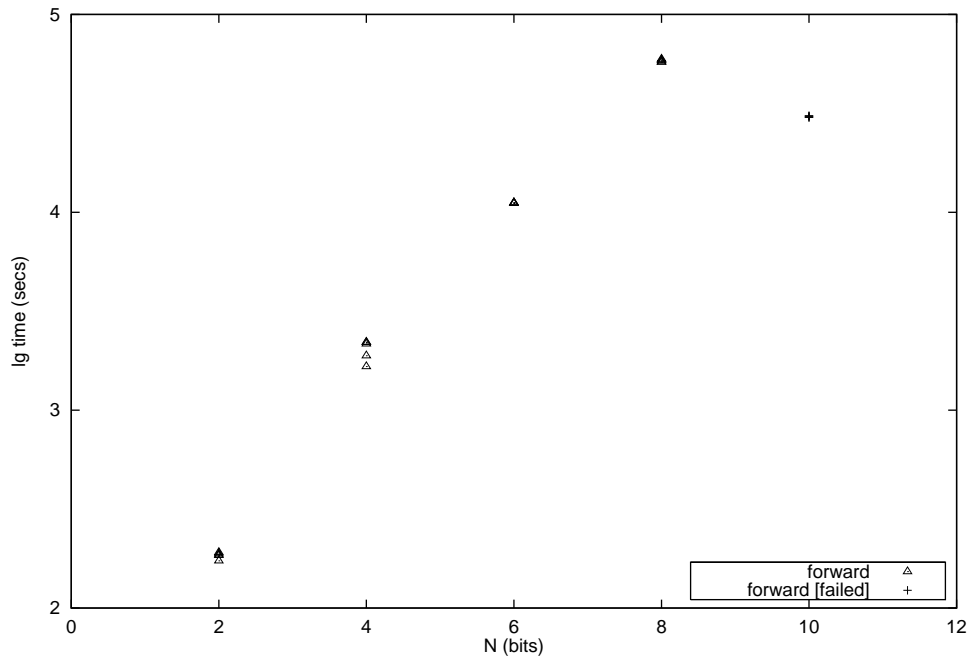


Figure 5.3: ASP Directional Performance: Time

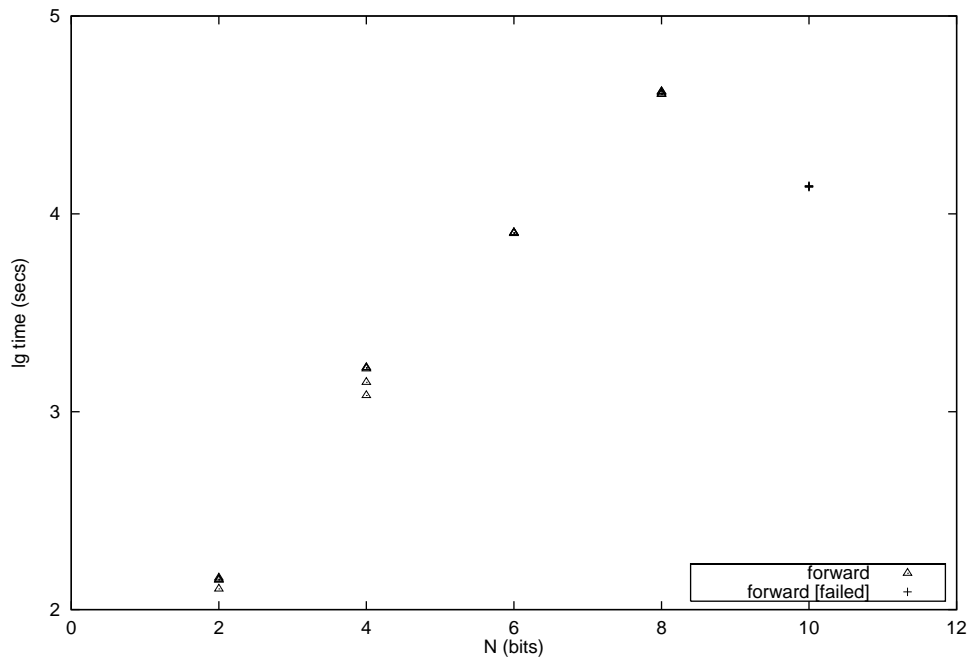


Figure 5.4: ASP Directional Performance: Search Time

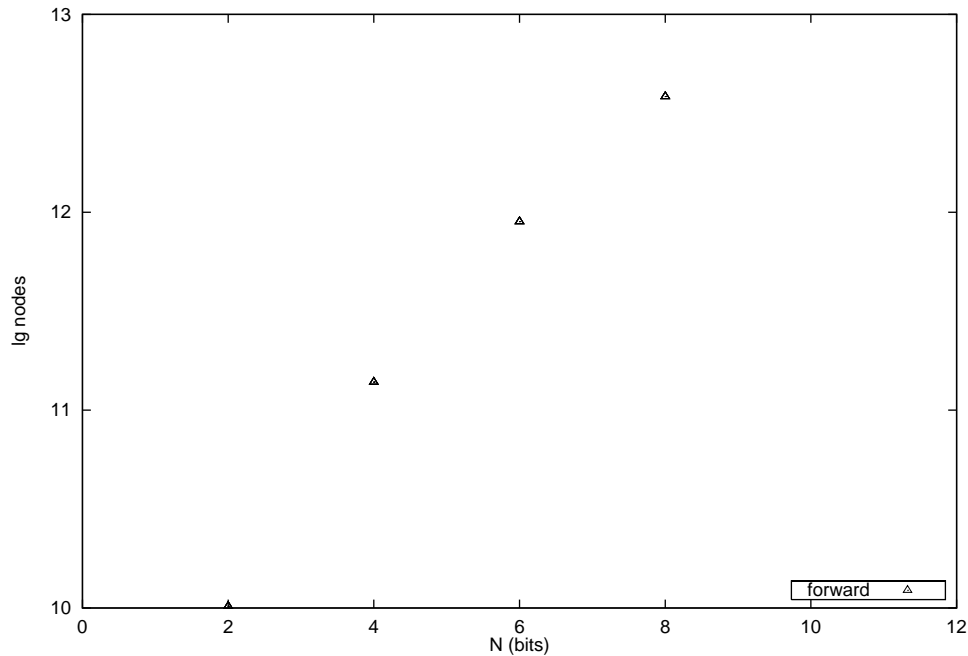


Figure 5.5: ASP Directional Performance: Nodes

5.2.3 Directionality of O-Plan

O-Plan is entirely a goal-regression planner, without even partial-order features; such a planner should perform well only on the backward problems. The results of figures 5.6 and 5.7 show that this is indeed the case. The times shown are wall-clock times for the entire planning process.

Note that O-Plan can solve only the $N = 2$ instance of the forward problem, while it is able to successfully negotiate the backward problem of size $N = 10$. In addition, note that the performance on the backward problem appears to be scaling linearly with increasing N . (The times reported for $N = 2$ are spuriously large, due to the 1-second granularity of the timer employed.)

The results reported here are for an older, slightly weakened version of the problem: O-Plan was unable to tractably plan in either direction when constraints were placed on the order of most actions (definition 4.16). In the actual implementation, only the nondeterministic input gate actions were ordered. Thus, while the results are plausible, they must be taken with a grain of salt at this point. It is intriguing that O-Plan and UCPOP, both previous-generation backward planners, have such difficulty with these apparently simple problems.

These experiments were conducted using a 3.1+ version of O-Plan based on the release of May 15, 1997, but with new code from 1998. Some minor modifications were necessary to make O-Plan run under Linux ACL, but these modifications, when executed on a Sun SPARCstation running SunOS 4.1.4, did not appear to in any way affect O-Plan's performance. To the best of the author's understanding, all of these modifications have been folded back into the release tree. The author thanks Jeff Dalton of AIAI in Edinburgh for his extensive help and hand-holding in getting this

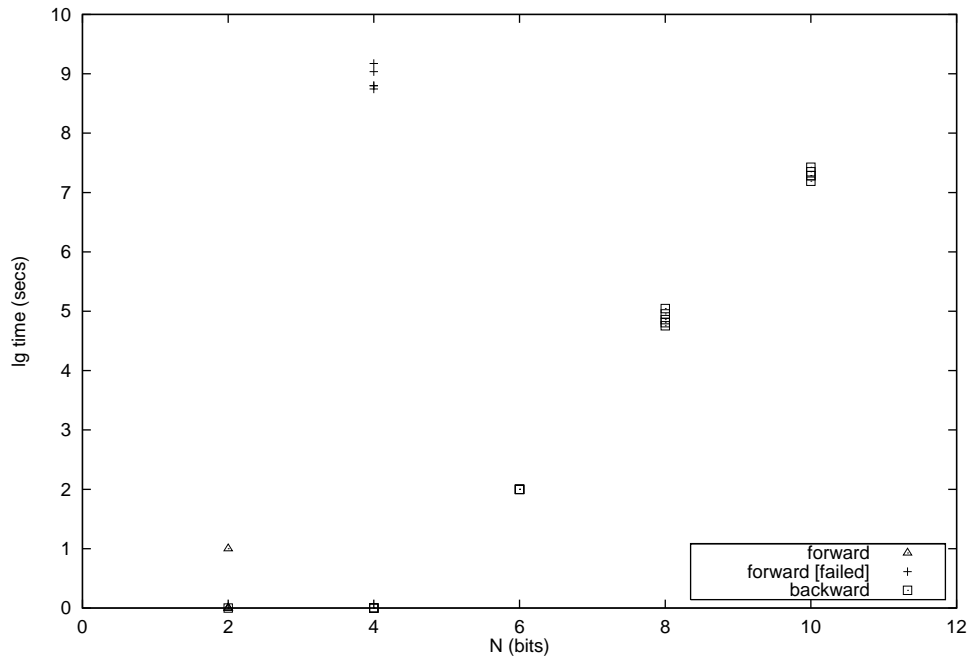


Figure 5.6: O-Plan Directional Performance: Time

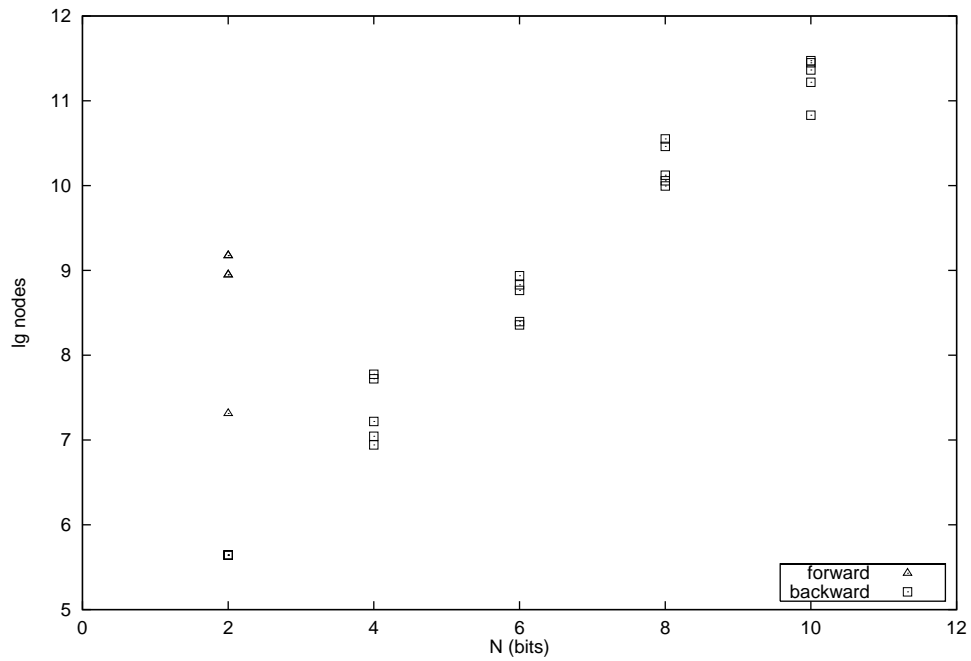


Figure 5.7: O-Plan Directional Performance: Nodes

planner integrated into the system, and Brian Drabble, currently at CIRL but formerly at AIAI, for his role in patiently answering numerous questions and providing the necessary contacts.

5.2.4 Directionality of UCPOP

The results of the directionality test applied to UCPOP are much less satisfying than the O-Plan results. As argued previously (p. 36) a partial order planner such as UCPOP, that adds actions to satisfy goal conditions and then regresses them, should plan tractably backward. Unfortunately, a key feature of UCPOP’s specific planning algorithm interacts with the construction of the problem to cripple UCPOP’s backward performance. As a result, UCPOP actually performs slightly better in the forward than in the backward direction, although the difference is not believed to be significant: at any rate, UCPOP cannot solve the $N = 6$ case, containing 114 operators, in either direction.

This is disappointing, since a great deal of effort has been devoted to producing a detection problem UCPOP can handle. That attempt has now been largely abandoned, after understanding a crucial property of UCPOP’s action selection mechanism. Imagine that there are two outstanding goals at a given point in partial plan construction, f and g , and two actions are available: action a that has effect $\{f\ g\}$, and action b that has effect $\{f\ \neg g\}$. Thus, action a will satisfy both goals. Since the plan under construction is partially ordered, it is possible that b could be part of a successful plan, but this will require that some action c subsequent to b achieve g . Unfortunately, UCPOP does not prefer a to b during search; further, it performs no check that any such c exists prior to inserting b into a partial plan. As a result, there is no way to force UCPOP to compute \mathcal{H}_n efficiently. O-Plan solves this problem by preferring a to b ,¹ which explains its ability to obtain good results.

Figures 5.8 and 5.9 show the only cases UCPOP appears to be capable of solving—increasing the time and search limits by an order of magnitude did not allow any $N = 6$ instances to be solved. The bottom line here is that, as explained previously (p. 49, p. 50), the fact that simple propagation from an arbitrarily selected state will quickly produce a plan does not mean that arbitrary search space planners can do so. UCPOP is one that cannot.

These experiments were conducted using UCPOP version 4.1, with parameters set to the defaults as shipped. The node counts represent the “plans explored” statistic reported by UCPOP. Exploratory experiments not reported here suggest that two other popular UCPOP flaw repair strategies, LCFR [26] and ZLIFO [52] (for a comparison of these strategies, see [43]), do not significantly change this result: neither addresses the question of action selection heuristics for filling open conditions. As UCPOP is flexible and modular, it would in principle be possible in principle to encode a custom flaw repair strategy that will select the desired open conditions; however, this would require some effort and would not in any case address the directionality of standard UCPOP configurations.

5.2.5 Directionality of Graphplan

Graphplan provides an interesting first test of the directional detector technique on a planner whose directional behavior is not completely understood. Graphplan is typically described by its creators as a forward planner, but its actual search mechanism is simple goal regression. The regression, however, is through a graph of actions and states that has been built up in a previous forward pass. This forward pass eliminates some states and actions that are impossible from the graph at graph

¹Brian Drabble, personal communication, 1998.

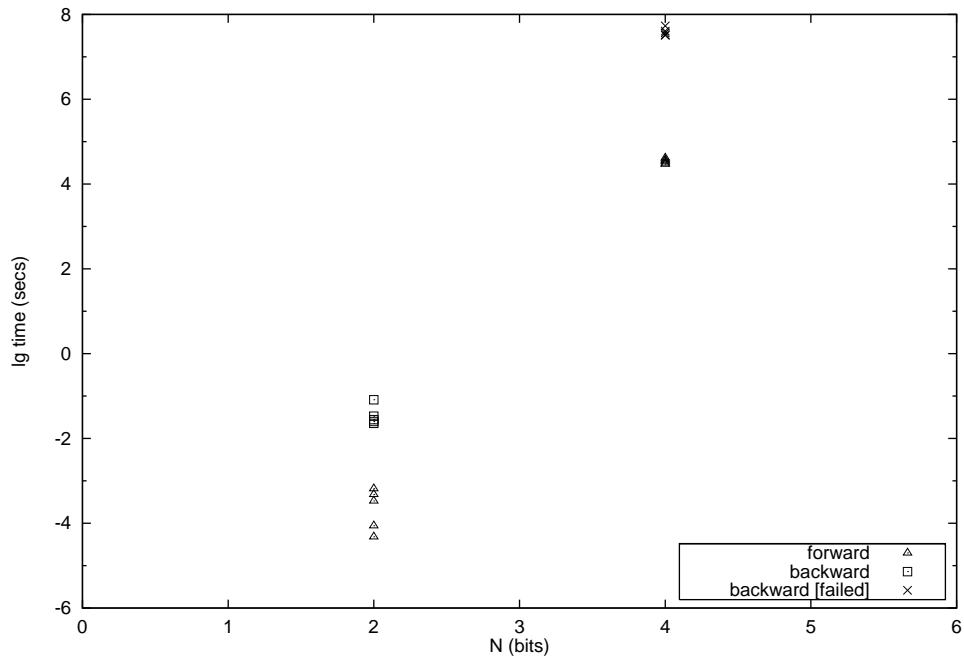


Figure 5.8: UCPOP Directional Performance: Time

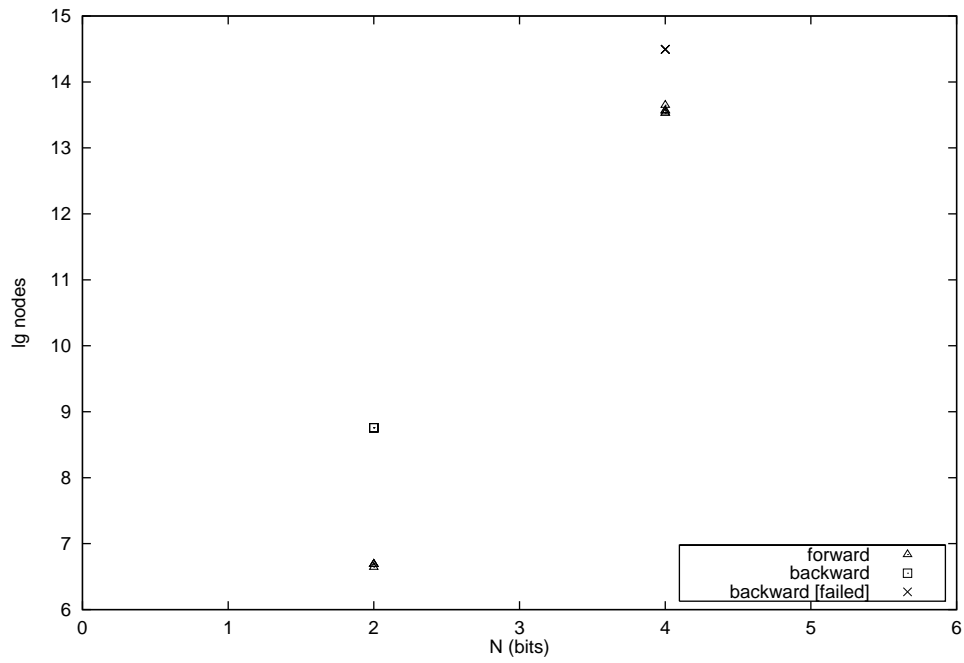


Figure 5.9: UCPOP Directional Performance: Nodes

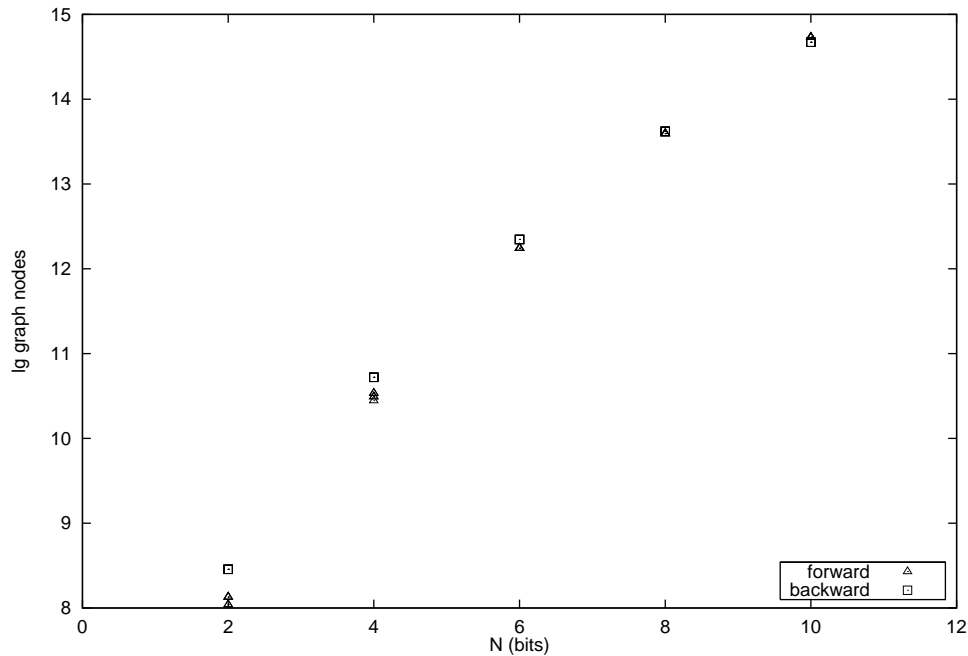


Figure 5.10: Graphplan Directional Performance: Graph Nodes

creation time, yielding a graph consisting of polynomially-sized layers that heavily prunes the backward search space.

The size of the plan graph in nodes (not to be confused with the search nodes of figure 5.12) as a function of N is given by figure 5.10. The forward and backward graph sizes appear to grow slightly sub-exponentially, as seen by the slightly sub-linear growth in the figure. The graphs should be polynomial in size, since the length of solution is polynomial. This polynomial will be high-order, however: since the length of the solution and the size of each layer are each $\mathcal{O}(n^2)$, the graph will be of size $\mathcal{O}(n^4)$ in general. The figure suggests that the forward graph may grow slightly faster than the backward graph with increasing N . This is to be expected, as the forward plan graph tries to capture increasingly large one-way functions, whereas the backward graph cannot capture much information about the one-way portion of the problem.

The overall performance graphs of figures 5.11 and 5.12 reflect that, as a search space planner (definition 4.2), Graphplan is bidirectional. The time performance shown in figure 5.11 reflects largely the cost of the graph construction. The plan graph is crucial in allowing efficient backward search on the forward problems. However, the graph construction itself begins to be a significant percentage of the total cost on the forward problems. Note that the search cost in nodes for the forward problems, shown in figure 5.12, becomes highly variable as N increases, most likely indicating the degree to which the plan graph construction is able to help search on a particular hash function. Nonetheless, the overall time cost, including the cost of graph construction, increases in an orderly and nearly linear fashion. The backward problems are solved in a straightforward fashion, resulting in a low search cost that is dominated by the cost of graph construction.

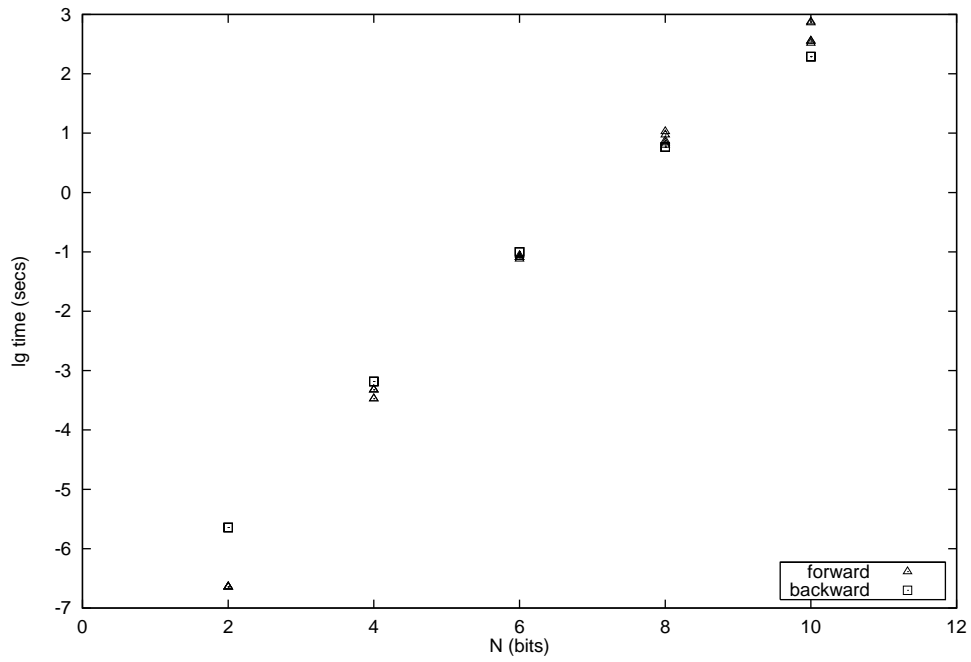


Figure 5.11: Graphplan Directional Performance: Time

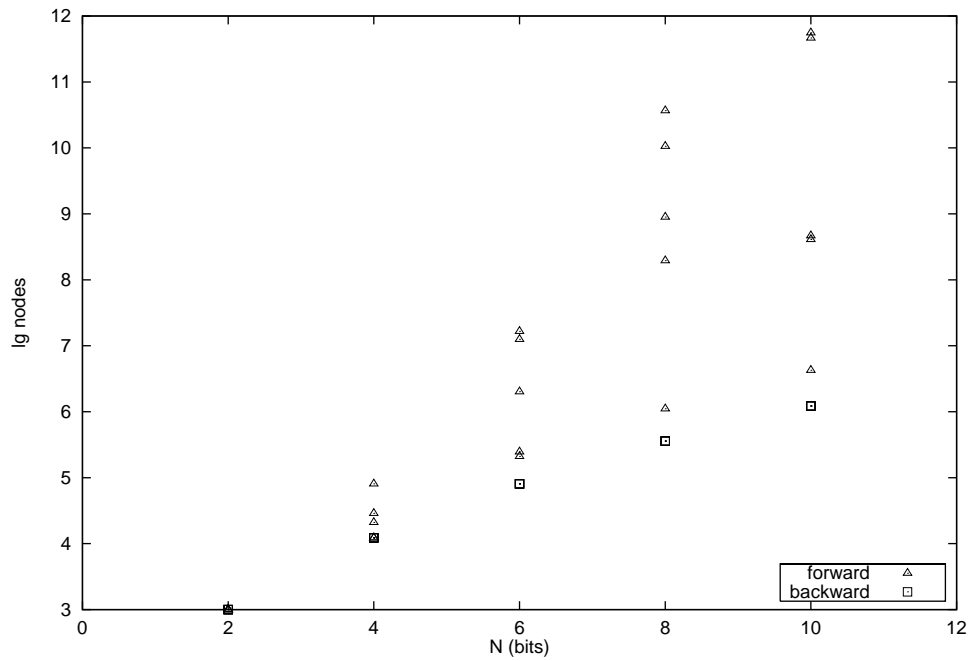


Figure 5.12: Graphplan Directional Performance: Search Nodes

These results illustrate some of the reasons why Graphplan was so inspiring to the research community when it was introduced. Its ability to adapt to different sorts of problems and its ability to handle large problems placed it far beyond any other fully general-purpose planner.

All experiments were conducted on the latest version of Graphplan (`graphplan.c` dated June 11, 1997) obtained by FTP from the Graphplan home page [8], using the defaults for all parameters except some of the memory sizes, which were increased to allow the larger experiments to be conducted. Minor modifications, not observed to affect performance, were made to allow imposing a search limit and to improve error reporting.

5.2.6 Directionality of Blackbox

The `blackbox` planner is the result of combining two fundamental successful ideas from recent planning research: the Graphplan approach described above, in which a graph is constructed to reduce the planning search space, and the “planning as satisfiability” approach of Kautz and Selman, as represented by their SATPLAN planner [28]. In this approach, a planning problem is transformed into a Boolean formula, and fast modern satisfiability engines are used to find a satisfying assignment for this formula. In `blackbox`, Kautz and Selman have found a way to use the plan graph to construct small, simple Boolean formulae, permitting quick discovery of a satisfying assignment.

Although some recent progress has been made in understanding search in satisfiability-based planners [48], it is still a matter of some conjecture how `blackbox` achieves its high levels of performance, and on what sorts of problems it will be effective. An understanding of the directional properties of the `blackbox` planning algorithm is therefore difficult to derive through understanding its inner workings. The detection technique described here, being an extrinsic test, is thus especially useful in answering these questions.

While it operates like Graphplan through the graph construction phase, `blackbox` offers a choice of solvers to proceed from this point. One possibility is for `blackbox` to emulate Graphplan to achieve a solution. More typically, `blackbox` can be asked to produce a satisfiability problem from the graph, and solve this problem using a variety of built-in solvers. This work focuses on two of these: the WSAT nonsystematic SAT engine, and Bayardo and Schrag’s `relsat` [6] systematic SAT engine. While Kautz and Selman report good results with the a rapid-restart approach based on Anbulagan and Li’s `SATZ` [2] engine, it was found to be highly inefficient in preliminary trials of both the forward and backward detector problems, and is not further considered here.

All measurements were made on `blackbox` version 3.2 using default settings, except for some of the memory sizes and time limits, that were increased to allow the larger experiments to run. All problems were given using PDDL encodings. Preliminary versions of these experiments were performed with `blackbox` version 1.0 in the Graphplan input format. Except for a slight improvement in speed, better robustness, and better error reporting, the newer version appears to be similar.

Blackbox Emulating Graphplan

The ability of `blackbox` to emulate Graphplan while running on problems encoded in a different input language provided a nice check on the experimental methodology. The graph sizes and search node counts were identical with those of Graphplan as depicted in figures 5.10 and 5.12, and the execution times, shown in figure 5.13, were not significantly different. This provided reassurance

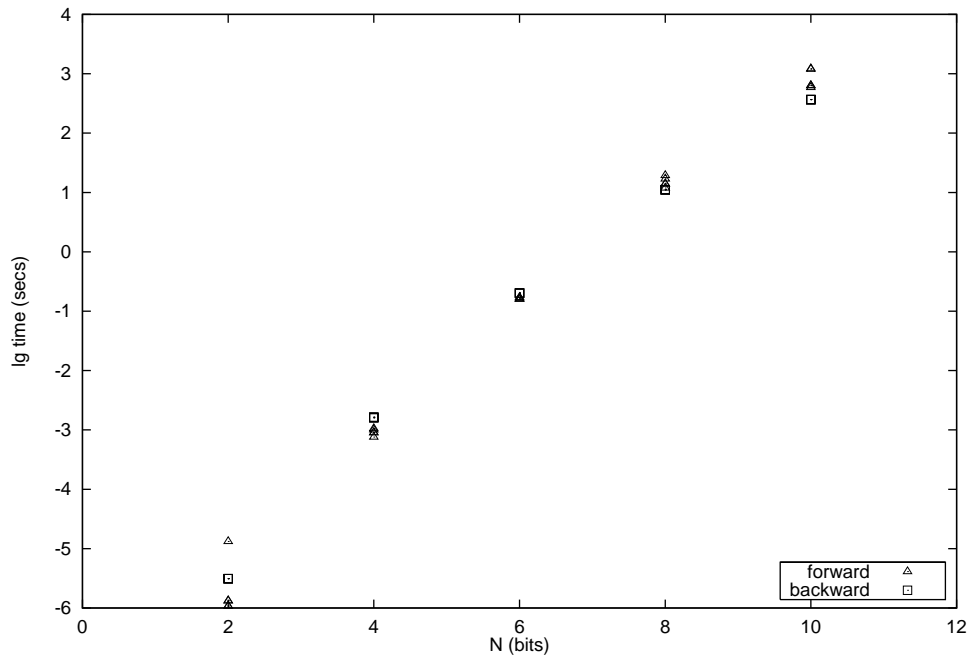


Figure 5.13: Blackbox/Graphplan Directional Performance: Time

both that `blackbox` faithfully implemented the Graphplan graph generation and that the problem encodings produced for these planners were functionally identical.

Blackbox Using WSAT

The execution times for `blackbox` with the WSAT solver, shown in figure 5.14, are disappointing. Essentially, these problems appear unsolvable by `blackbox` using WSAT. Perhaps tuning of the many parameters of WSAT would improve performance. However, experience with simpler problems² suggests that WSAT may be generally unsuitable for solving planning problems requiring large plans.

Blackbox Using Relsat

The `relsat` solver [6] is currently far and away the best systematic SAT solver available, and is competitive with WSAT on most structured problems. The execution times for `blackbox` with `relsat`, shown in figure 5.15, reflect its superiority on these instances. Over the measured range of problem sizes, performance of `relsat` as a search engine was quite acceptable, although the simple backward search used in Graphplan appears to be superior. Note that the graphs show problems only up to $N = 8$. Beyond this size, problem encodings were too large to fit in physical memory (approximately 200MB), causing unacceptable swapping during search.

²Andrew Parkes, using tools the author helped develop to explore the behavior of WSAT on SAT encodings of logistics planning problems, has observed that WSAT tends to get stuck in local minima on planning problems requiring long plans.

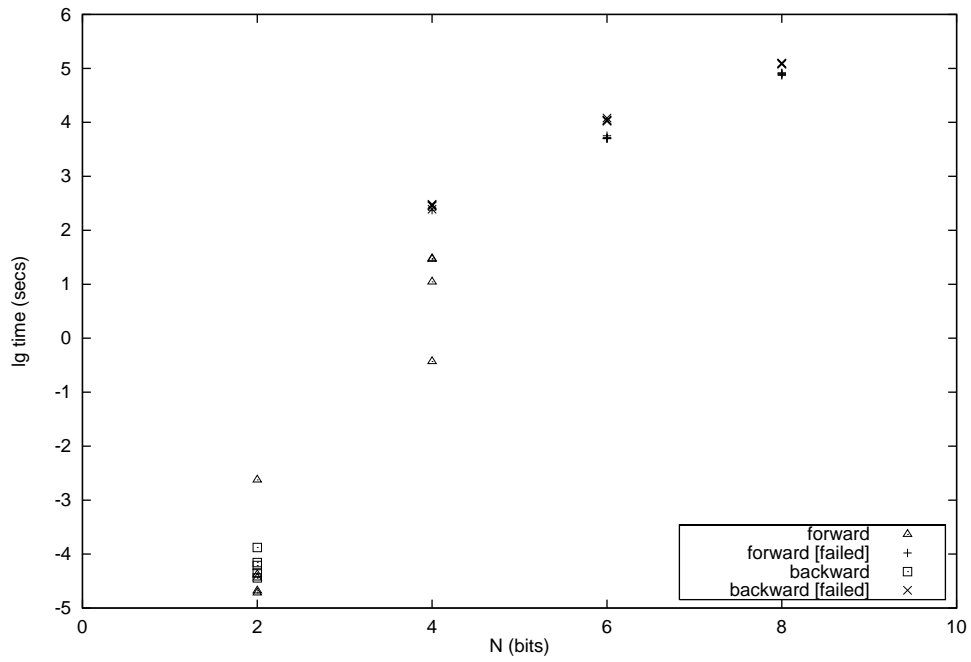


Figure 5.14: Blackbox/WSAT Directional Performance: Time

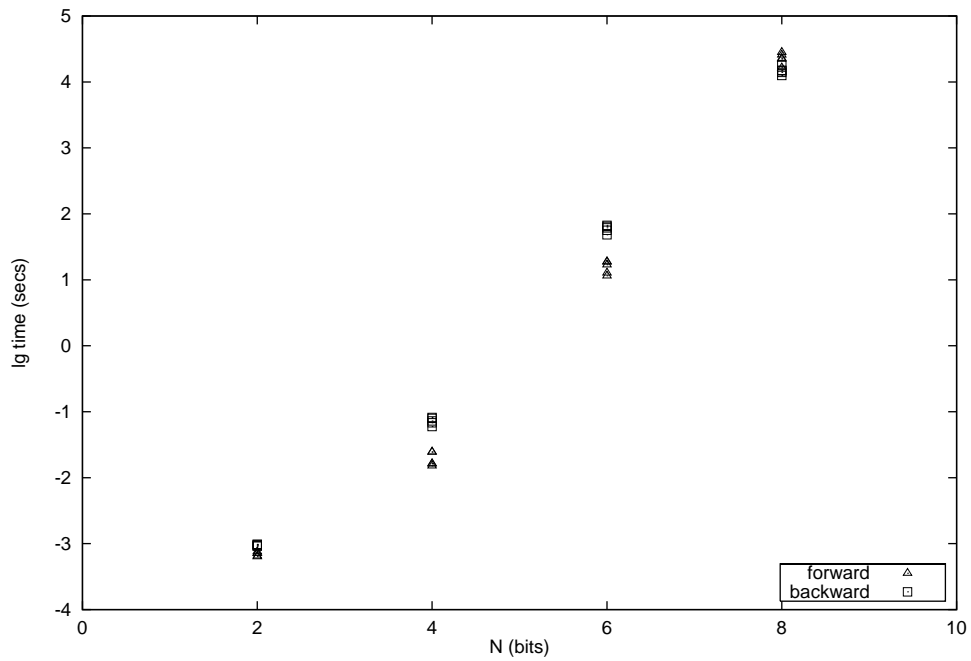


Figure 5.15: Blackbox/Relsat Directional Performance: Time

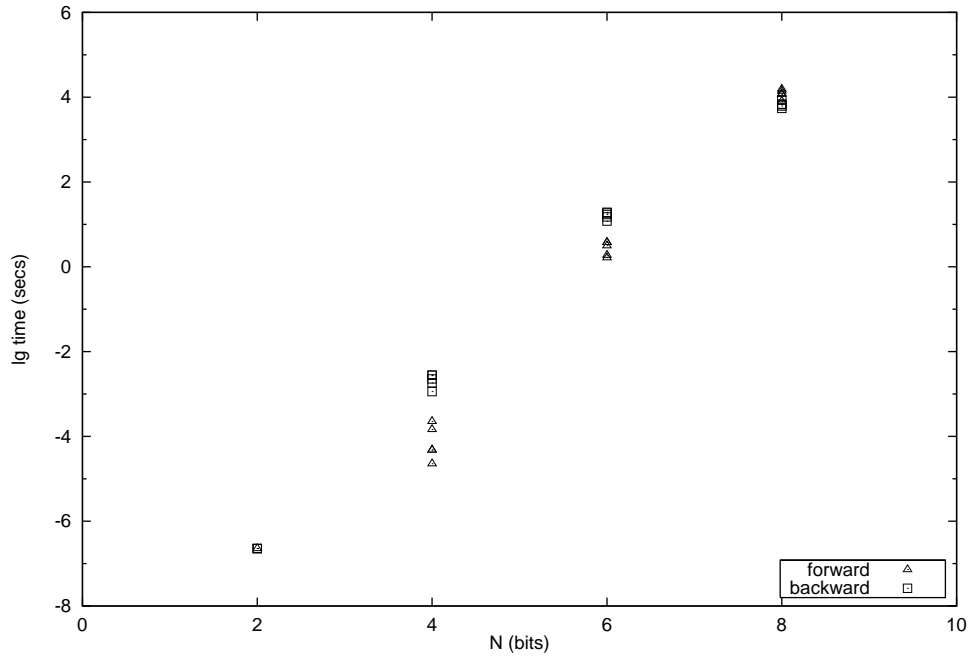


Figure 5.16: Blackbox/Relsat Directional Performance: Relsat Time

In separating out the component of solution cost due to search, it is apparent that, if anything, the forward problems are easier for the relsat implementation. The search time, shown in figure 5.16, is small, though roughly exponential. The search cost in nodes, shown in figure 5.17, is similar. Here, nodes searched is represented by the “variables valued” statistic of relsat.

5.3 Parallel Plans

The performance of Graphplan and the blackbox variants on the detector problems is slightly disappointing. These planners generally are quite good at solving problems significantly larger than the detector problems. Paradoxically, an important constraint in the performance of these planners on these problems is the condition that actions occur in sequential order. Planners that can construct *parallel plans*, in which non-interacting actions may occur simultaneously, can search to greater depth in a given time, and can construct temporally shorter plans.

The constraints that sequentialize actions in NBCPPs are the do_i preconditions of definitions 4.16 and 4.17. These constraints also prevent a given gate from being evaluated more than once, though, and thus cannot be simply discarded. Instead, they can be replaced by constraints that delete the inputs of a gate once the gate has been evaluated. The following definitions capture this notion.

Definition 5.2 (Negative-Signed Formula) For any conjunctive formula F (definition 1.3), the negative-signed formula $F^{\bar{s}}$ is given by

$$F^{\bar{s}} = \{\neg f^+, f^- \mid f \in F\} \cup \{f^+, \neg f^- \mid \neg f \in F\}$$

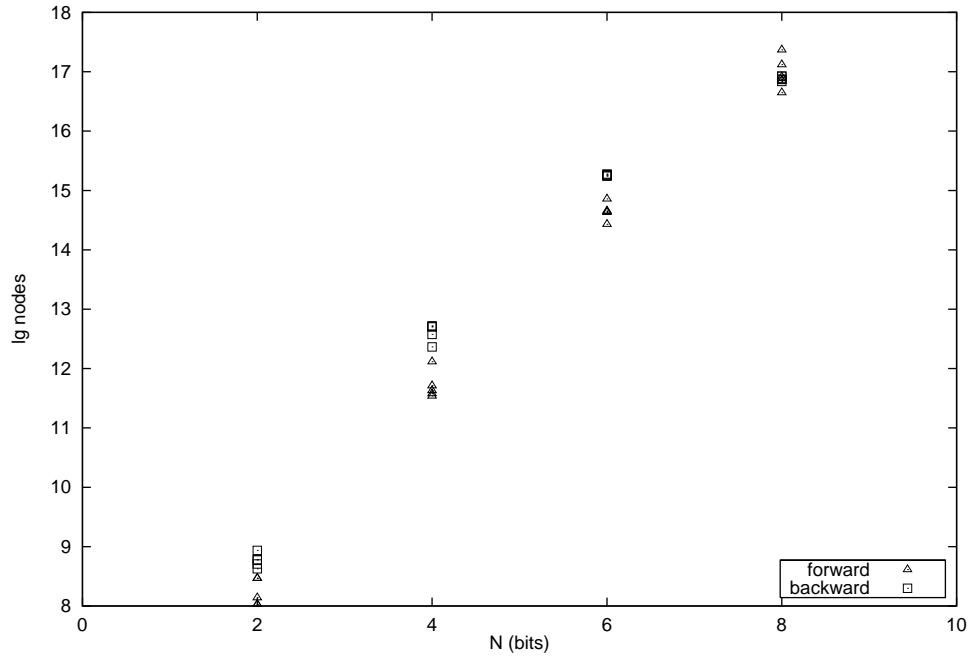


Figure 5.17: Blackbox/Relsat Directional Performance: Variables Valued

Definition 5.3 (Parallel NBCPP) For any Boolean circuit $C = \langle V, E \rangle$, the parallel NBC planning problem $\langle \langle F, A \rangle, I, G \rangle$ is constructed as follows: Define the fluents of the problem by $F = V^\pm(C)$. Define the initial state of the problem by $I = \text{in}^s(C)$, and the goal state of the problem by $G = \text{out}^s(C)$.

Given conjunctive formulae α and β , let the action $a_{\alpha\beta}$ that requires α and produces β be defined by

$$a_{\alpha\beta} = \frac{\alpha^s}{\beta^s \alpha^{\bar{s}}}$$

Then given a gate v with truth table T_v , and an input $\alpha \in \text{dom}(T_v)$, the set of actions $A_\alpha(v)$ associated with input α is

$$A_\alpha(v) = \{a_{\alpha\beta} \mid \beta \in T_v(\alpha)\}$$

Then the set of actions A for the entire domain is just the set of actions for all inputs of all gates.

$$A = \bigcup_{v \in V, \alpha \in \text{dom}(T_v)} A_\alpha(v)$$

Definition 5.4 (Parallel Reverse NBCPP) The parallel reverse NBC planning problem associated with a circuit C is constructed by reversing the initial and goal states and the actions of the parallel NBCPP for C . Given definition 5.3, define instead the initial state of the problem by $I = \text{out}^s(C)$ (see definition 3.2), and the goal state of the problem by $G = \text{in}^s(C)$.

Given conjunctive formulae α and β , let the action $\hat{a}_{\alpha\beta}$ be defined by

$$\hat{a}_{\alpha\beta} = \frac{\beta^s}{\alpha^s \beta^{\bar{s}}}$$

Then given a gate v with truth table T_v , and an input $\alpha \in \text{dom}(T_v)$, the set of actions $\hat{A}_\alpha(v)$ associated with input α is

$$\hat{A}_\alpha(v) = \{\hat{a}_{\alpha\beta} \mid \beta \in T_v(\alpha)\}$$

Then the set of actions A for the entire domain is just the set of actions for all inputs of all gates.

$$A = \bigcup_{v \in V, \alpha \in \text{dom}(T_v)} \hat{A}_\alpha(v)$$

The proof of correctness is similar to lemma 4.2. Only the proof for the forward problem of definition 5.3 will be given here: the proof for definition 5.4 is similar.

Lemma 5.1 (NBC Evaluation Parallel Planning) *Given a NBC C and the corresponding parallel NBCPP $P(C)$, a plan ρ solving $P(C)$ exists if and only if C computes the given outputs from the given inputs. Further, if any ρ exists, there exists a ρ such that $|\rho| = |C|$ (the length of the plan is the same as the number of gates in the circuit).*

PROOF: By dual implication.

Circuit solution \implies plan existence:

Consider the input values α and output values β at any gate $v \in C$, and the corresponding set of actions $A_\alpha(v)$ (definition 4.16). By construction, there must be an action $a_{\alpha\beta} \in A_\alpha(v)$ (and therefore $a \in A$) with preconditions α^s and effects $\beta^s \cup \alpha^{\bar{s}}$. Thus, a plan can be constructed as follows: Take any topological sort v_1, v_2, \dots, v_n of the vertices of C , and begin with the empty plan ρ_0 . Now, for each vertex v_i of C in turn, append action $a(v_i)$ to ρ_{i-1} to obtain ρ_i . Since each of the inputs of each vertex v_i must be drawn only from the outputs of earlier vertices v_1, v_2, \dots, v_{i-1} , the input fluent values α of $a_{\alpha\beta}(v_i)$ are fixed. Thus, the action with output β can be chosen. Finally, by our earlier observation, the plan will produce the required goal state fluents, so ρ is valid, and $|\rho| = |V|$.

Plan existence \implies circuit solution:

Consider the preconditions and effects of any action $a_{\alpha\beta}(v)$ in ρ . By definition, a corresponding gate v is in C , and given input α can produce output β . Thus, the circuit can be evaluated as follows: The last action $a_n = a_{\alpha\beta}(v)$ of ρ must produce only goal conditions, and must correspond to the appropriate evaluation of gate v_n of C . Set the inputs and outputs of v_n according to α and β . Now, consider each action $a_{\alpha\beta}$ producing a precondition of a_n . Each must correspond to a gate v whose output is an input of v_n . Set the inputs of v to α , and its outputs to β . Continue until all values of all gates of C have been set. This must happen, since the fluents of P have been uniquely labeled according to the signals of C . Thus, the evaluation of C is correct.

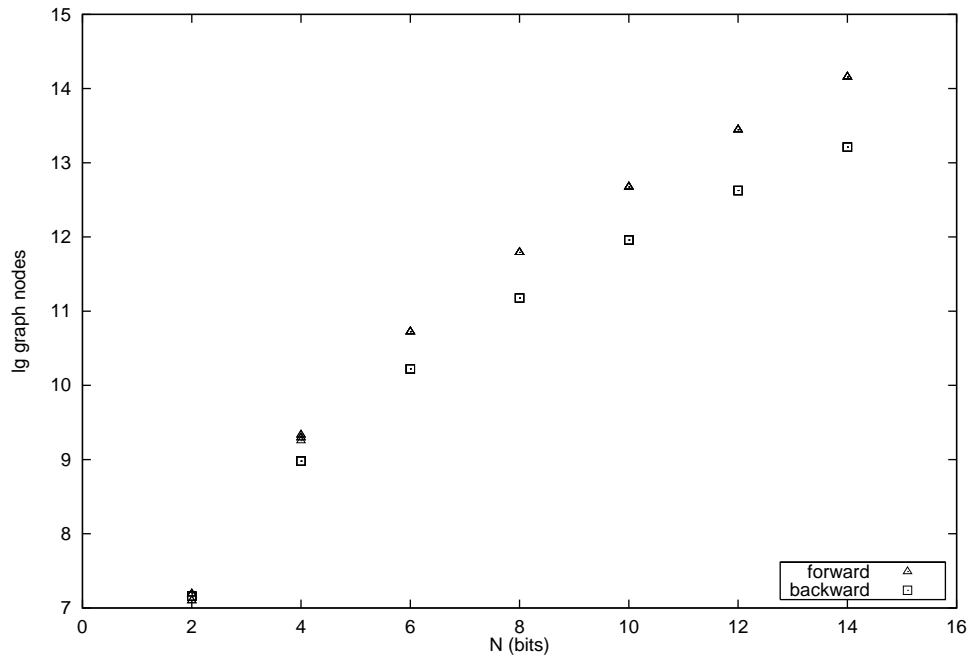


Figure 5.18: Parallel Graphplan Directional Performance: Graph Nodes

Note that there may exist many plans corresponding to a given circuit evaluation, corresponding to different topological sorts of the circuit.

These parallel planning problems have been implemented for the planning systems described previously: the results are quite consistent with those of the previous section, but the ability to experiment with larger problem sizes provides a nice confirmation of the data.

5.3.1 Graphplan

The expected polynomial for the graph size is now $O(n^3)$, since the expected length of a plan is now $O(n)$. Indeed, the sub-linearity of figure 5.18 is more pronounced than in the sequential case. Overall, the graphs are similar in character to the sequential case. In figure 5.19, and especially in figure 5.20, the variability on the forward problems increases rapidly with increasing N . Finally, note the impressive size of the problems solved: when $N = 14$, these problems have 706 operators.

The performance of `blackbox` emulating Graphplan is nearly identical, and is not shown here.

5.3.2 Blackbox Using WSAT

The parallel problems enable evaluation of `blackbox` with the WSAT solver, by reducing the encodings to manageable size. The running times, shown in figure 5.21, are interesting: they are similar in character to the Graphplan results, but with somewhat worse absolute performance. In particular, solving the larger forward detector problems with WSAT appears to result both in longer run times and in higher variance in the runtime. While the latter result was expected, the former is

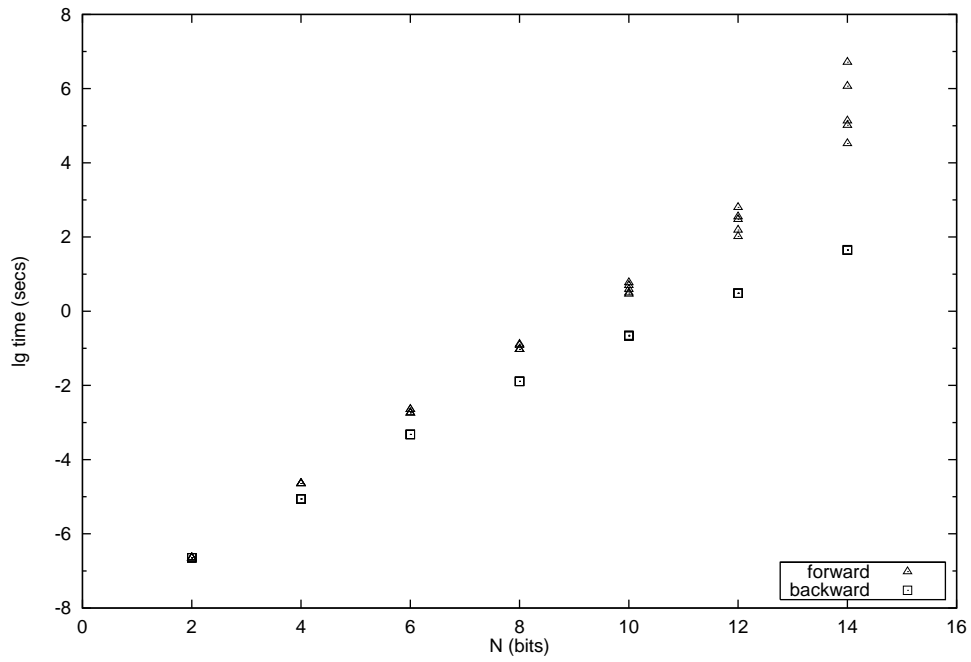


Figure 5.19: Parallel Graphplan Directional Performance: Time

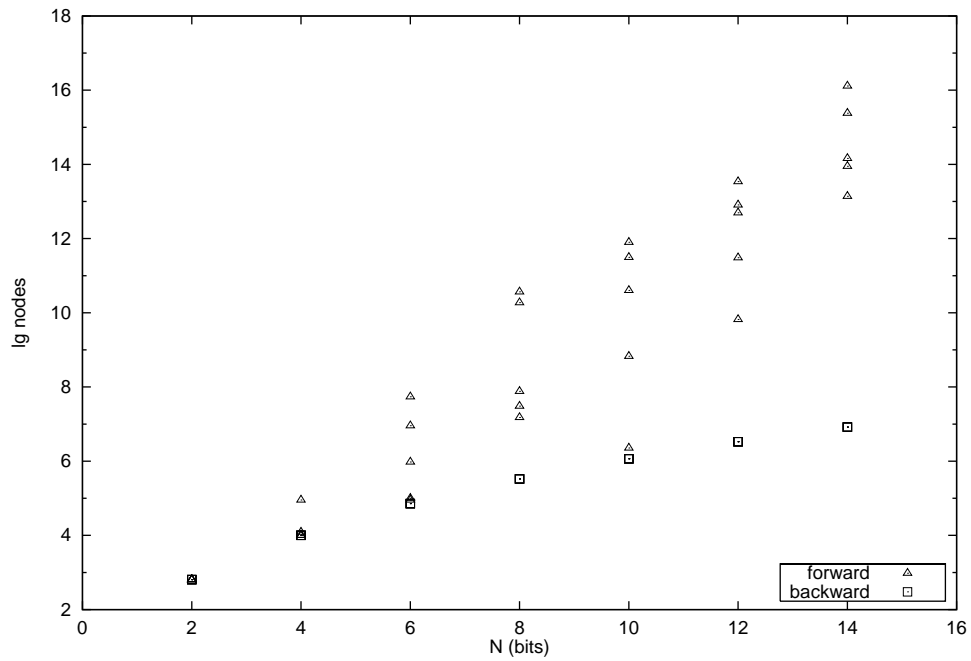


Figure 5.20: Parallel Graphplan Directional Performance: Search Nodes

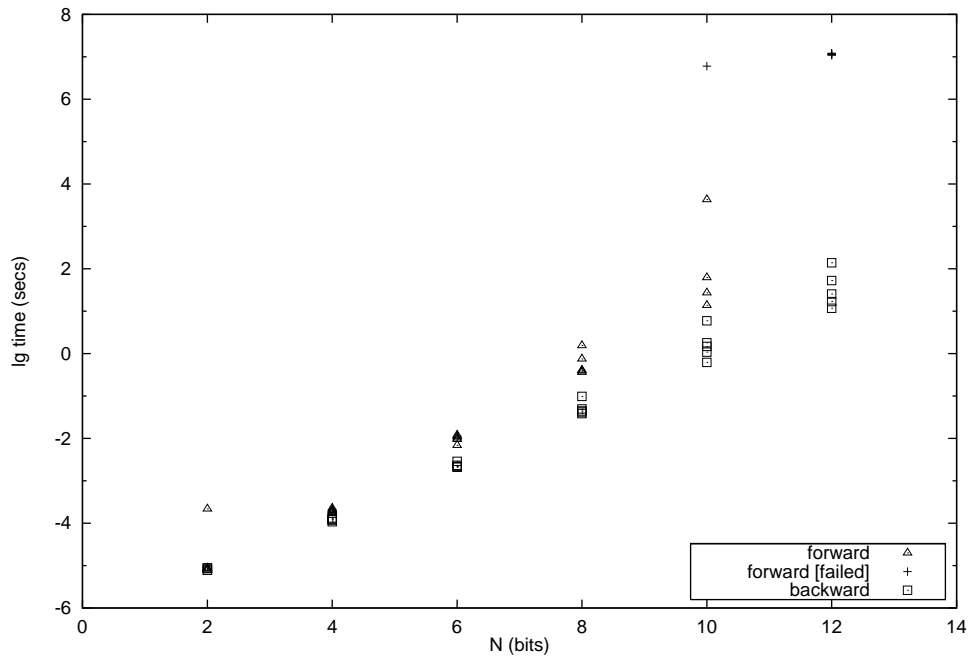


Figure 5.21: Parallel Blackbox/WSAT Directional Performance: Time

surprising, inasmuch as satisfiability-based solvers such SATPLAN and Medic using WSAT have been reported to be competitive with Graphplan on a variety of problems.

Figure 5.22 shows the time consumed by WSAT for the solved instances. Figure 5.23 shows the WSAT flip count, effectively the number of search nodes to WSAT solution. While the forward and backward plots are similar, and the variances in both directions are high, there is still a noticeable backward bias to the search. The author speculates that this may be the result of bias in the satisfiability encoding used by `blackbox`. This encoding was inspired partly by a study of Graphplan, and might perhaps be expected to be reverse biased. It is also possible that the underlying axioms used in the encoding are directionally biased, although this would be more surprising. In any case `blackbox`/WSAT appears to behave as a bidirectional planner with a slight backward bias.

5.3.3 Blackbox Using Relsat

The execution times for `blackbox` with `relsat` on the parallel problems, shown in figure 5.24, are quite impressive. By comparison with the WSAT times of figure 5.21, these times are again similar in character: the times are much shorter, however, and show little variance.

The search time, shown in figure 5.25, is extremely small, though roughly exponential once it grows large enough to measure. The behavior in nodes searched, shown in figure 5.26, is interesting. Here, nodes searched is represented by the “variables valued” statistic of `relsat`: in all of the forward $N = 8, 10, 12$ cases tried, all variables of the problem were valued once. This indicates the ease with which `relsat` solves these problems.

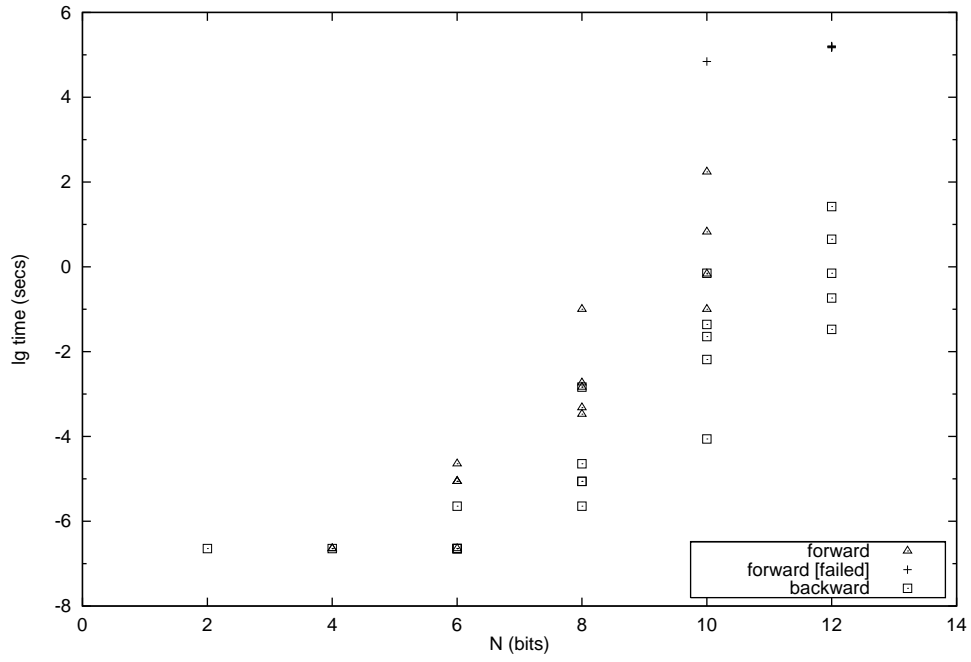


Figure 5.22: Parallel Blackbox/WSAT Directional Performance: WSAT Time

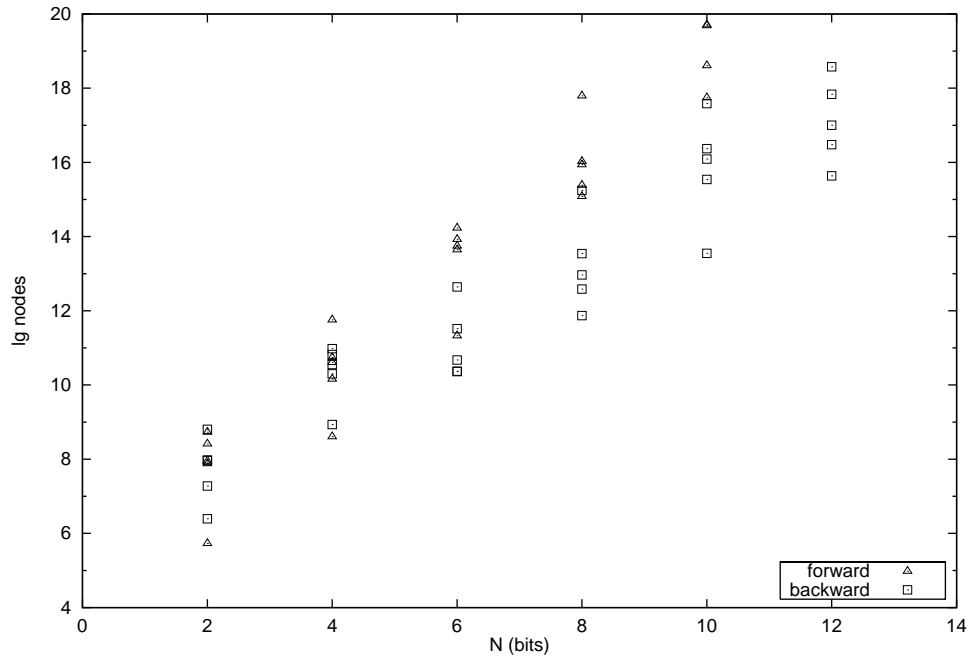


Figure 5.23: Parallel Blackbox/WSAT Directional Performance: WSAT Flips

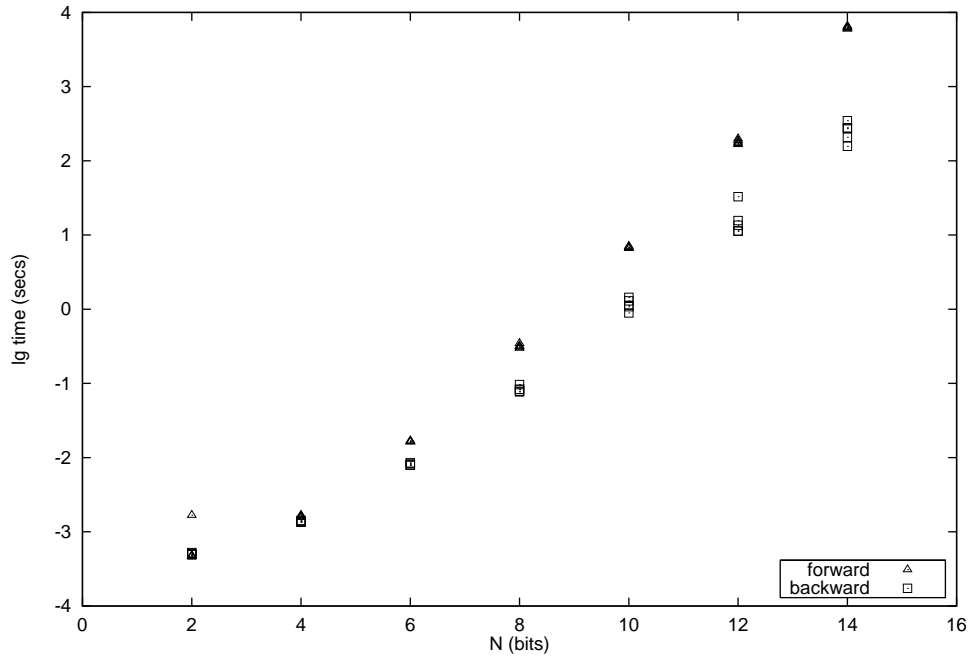


Figure 5.24: Parallel Blackbox/Relsat Directional Performance: Time

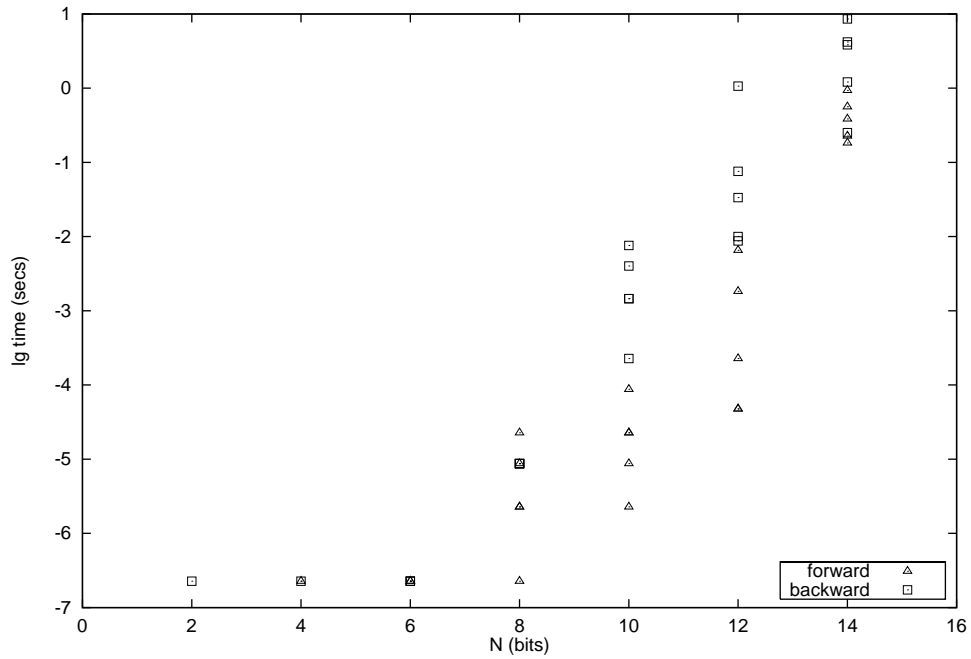


Figure 5.25: Parallel Blackbox/Relsat Directional Performance: Relsat Time

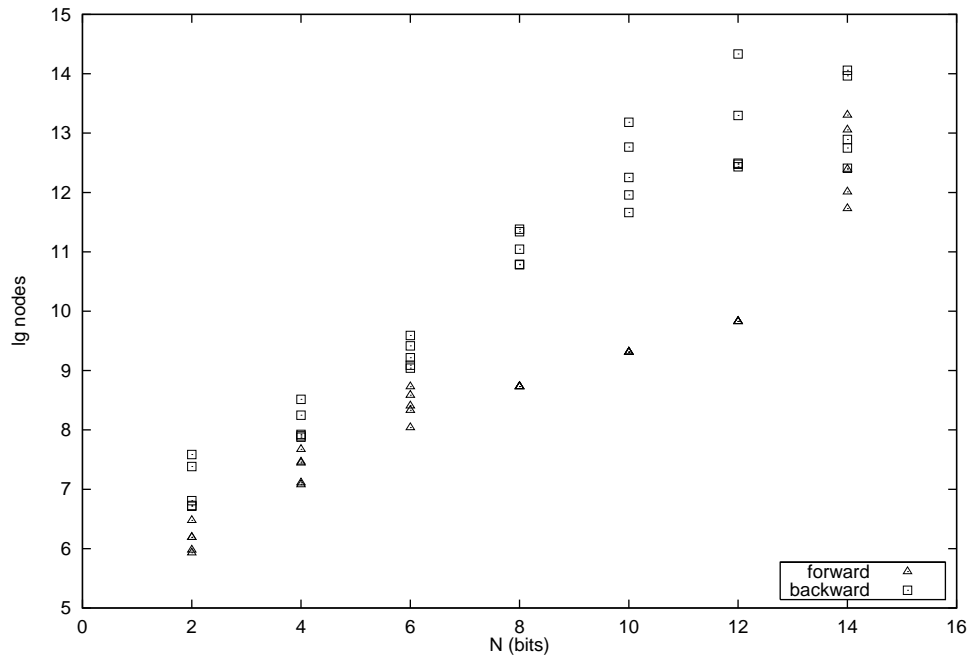


Figure 5.26: Parallel Blackbox/Relsat Directional Performance: Variables Valued

5.4 Overall Results

The experimental results of this chapter can be summarized as follows:

- ASP is a forward planner. This is not a surprising conclusion, but does provide a validation of the methodology.
- O-Plan appears to be a backward planner.
- UCPOP is too inefficient to allow conclusions to be drawn about its directionality using this methodology.
- Graphplan is a bidirectional planner.
- Blackbox has directional characteristics similar to those of Graphplan, although the SAT solvers themselves vary slightly in both performance and directionality.

Chapter 4 began with three questions:

1. How should the directionality of a planning algorithm be *defined*?
2. How can the directionality of a planning algorithm as defined in question (1) be *determined*?
3. How does the directionality test of question (2) work out in practice?

The answer to question (1) was given by a reasonable formal definition of search based planning. The answer to question (2) was given by a theoretically sound extrinsic technique for determining planner directionality. In this chapter, the answer to question (3) was given by a series of planner directionality experiments, permitting some insight into the operation of poorly understood modern planners.

Chapter 6

Directions In Planning

The necessary ingredient in achieving the results of Chapters 3–5 is the discovery of a reencoding of a problem as a PROPS problem. In chapter 3, a reversed problem is compiled from a somewhat richer language into a PROPS problem. In chapter 4, a one-way hash function is encoded as a PROPS problem.

In this chapter, some issues related to these reencodings are discussed, with reference to their bearing on planning directionality. This leads naturally to a discussion of the expressive power of STRIPS, and its bearing on directionality. The chapter concludes with a discussion of the impact of this work on existing planning algorithms, and some general conclusions.

6.1 “Don’t Know” and “Don’t Care” in PROPS Actions

The compilation schemes given in chapter 3 are sufficient for the purposes of the reversal algorithm. However, the notion of DC effect presented there (pp. 25–26) can be generalized in a natural fashion. The resulting language is perhaps more satisfying from an intellectual point of view; its extra expressiveness is of a sort that may prove to be more useful than standard PROPS for encoding real world problems.

6.1.1 Unrestricted DC Effects

In chapter 3, the notion of a “Don’t Care” or DC effect was described, in the context of a compilation scheme C_3 taking a PROPS variant containing DC effects to an isomorphic problem in standard PROPS. A DC effect is an effect that can satisfy either positive or negative preconditions of an action: a sort of wild-card effect.

DC effects were implemented in the compilation scheme C_3 as a pair of effects setting both the positively and negatively signed fluents while reencoding the problem into one with only positive preconditions. The DC effects were restricted to occur in actions only for fluents that were mentioned as a precondition of the action.

The reason for the restriction, not discussed in chapter 3, is that this is all that is necessary for the reversal. In the reversal rules (figure 3.3), a DC effect appears in a reversed problem only where a fluent was mentioned in the effects of an action in the original problem. Thus, the only reason to

generalize to unrestricted DC effects is to permit explicit DC effects in problem encodings. As will be seen shortly, this approach has its own features and pitfalls.

6.1.2 DC Effects and the Initial State

It would seem natural that explicit DC effects be allowed in the initial state of problem descriptions. The discussion of the reversal of problems with partial goal states (chapter 3, p. 30) explicitly describes how to compile out DC effects in the initial conditions through a simple extension to C_3 , since this is necessary to handle the initial states obtained by reversing problems with partial goal states using C_R .

Allowing DC effects to be explicitly given in the initial conditions when encoding problems in PROPS has several advantages. For one thing, the initial state is often treated in the literature as the effect of an initial action, for the purpose of simplifying proofs: if DC action effects are permitted, then DC initial conditions should be as well.

More importantly, explicit DC initial conditions would allow easy expression of the notion that, for a particular problem, actions in the domain should be unrestricted by some of the initial conditions. This is a notion that is sometimes useful in real-world problems.

Consider, for example, a problem domain that has some actions that are executable only on even days, and some that are executable only on odd days. One obvious PROPS encoding D_1 has two fluents, *odd-day* and *even-day*, one of which would normally be set in the initial conditions. Another obvious encoding D_2 involves only an *odd-day* fluent, that is set either true or false initially. Neither of these domain encodings, however, make it convenient to pose the planning problem “find a plan for this problem, for any particular day.” If both fluents are set true in the initial state of the problem using the D_1 encoding, *odd-day* and *even-day* actions may be mixed in the resulting plan. Things are even worse for the D_2 encoding: there is no obvious way to express the problem at all.

In the D_1 encoding, an action can be added to the domain that can only be executed once and has the effect of making the day either odd or even. This is uncomfortable, both because it seems inelegant to modify the domain in order to handle a particular problem, and because many planners fail to scale well as additional actions are added to domains. In the D_2 encoding, it appears that the best that can be done is to pose two planning problems, one in which the actions for odd days are enabled, and one in which the actions for even days are. Unfortunately, if there are many such fluents to be considered, this leads to a combinatorial explosion.

Fortunately, a problem encoding based on D_2 , with an initial DC value for the fluent *odd-day*, allows the desired problem to be posed directly. Because of the treatment of DC values described in chapter 3, a plan for this problem may contain *odd-day* actions, or \neg *odd-day* actions, but never both. This notion is intuitively reasonable, simple to implement (through compilation using C_3), and still permits problem reversal using C_r .

Note that DC initial conditions and actions with DC effects are different than the anomalous actions due to ambiguity in the situational calculus described by Manna and Waldinger [32]. They note that plans expressed in a standard formulation of the situation calculus permit actions which are not executable in the real world.

For example, consider [a problem in which] a monkey is presented with two boxes and is informed that one box contains a banana and the other a bomb, but is not told which.

His goal is to get the banana, but if he goes anywhere near the bomb it will explode. As stated, the problem should have no solution.

Manna and Waldinger go on to point out that theorem proving in the standard situational calculus yields a plan in which the monkey takes an action roughly equivalent to “go to the box such that it would get the banana if it went there.”

This sort of non-executable action seems to indicate a problem with the standard situational calculus (for which Manna and Waldinger suggest a solution). However, DC initial conditions and effects are of a different nature: they are still deterministic and require known preconditions. Indeed, since DC initial conditions and effects are expressible in PROPS, they add no real (i.e. logical) power to planning. Thus, the sort of problem described by Manna and Waldinger never arises in this context.

6.1.3 Explicit DC Action Effects

Having admitted the notion of explicit DC values in the initial state, the next logical question is whether to permit explicit DC values in operator effects (that is, DC effects given directly in the problem encoding, rather than as a result of reversal as in chapter 3). Three questions that must be addressed are:

1. How would explicit DC effect semantics be useful in real-world encodings?
2. How would explicit DC effect semantics be implemented using traditional planners?
3. Are domains or problems containing explicit DC effects reversible?

In considering question (1), it is difficult to think of situations in which an effect of an action should be to make a fluent “true or false, whichever is needed.” As explained above, this does not include the sort of nondeterminism that allows actions such as “dump the water into the hole with the fire, even though I’m not sure which hole that is,” as this sort of action is not expressible in PROPS even using DC effects. Generally, an action effect is encoded into a domain description either because the effect is useful, or because it is unavoidable (a *side-effect*). In either case, the effect is almost invariably specific, either setting or clearing a fluent. Thus, there is little motivation to introduce explicit DC action effects into domain encodings.

As to question (2), it is easy to see how to extend compilation scheme C_3 to handle unrestricted explicit DC action effects: the compilation scheme C_{*3} for unrestricted DC effects consists of all the rules of C_3 plus the compilation rule

$$\frac{}{*f} \Rightarrow \frac{}{f^+ f^-}$$

Finally, the answer to question (3) points up a real problem with unrestricted DC effects. The reversal rules of figure 3.3 are written as unidirectional rules, to simplify the presentation. It is notable that the first four rules are manifestly symmetric: they can be read from right to left, with rules 3.9 and 3.10 remaining the same, and rules 3.11 and 3.12 interchanged. Rules 3.13 and 3.14

$$\frac{f}{*f} \xLeftrightarrow{\quad} \frac{-}{f} \quad (6.1)$$

$$\frac{\neg f}{*f} \xLeftrightarrow{\quad} \frac{-}{\neg f} \quad (6.2)$$

Figure 6.1: Reversal Rules for Restricted DC Effects

are more problematic, but an inspection of the proof of correctness of C_R shows that these rules may be read from right to left as well, giving two new rules shown in figure 6.1.

The rules of figure 6.1 cover the case of reversal of explicit restricted DC effects, but the unrestricted case is more problematic. The obvious approach is by analogy with rule 6.1: Delete the fluent f from this rule, yielding the rule

$$\frac{-}{*f} \xLeftrightarrow{\quad} -$$

Unfortunately, this rule can be easily seen to be incomplete, by considering a problem with initial state $\{f\}$, goal state $\{\neg f\}$, and sole operator

$$\frac{-}{*f}$$

The single-action plan is then legal in the forward domain, but its reversal is illegal in the reversed domain. (One might suspect trouble in any case, since reading this rule right-to-left yields the rule

$$- \xLeftrightarrow{\quad} \frac{-}{*f}$$

which is obviously nonsensical.)

An operator containing an unrestricted DC effect can be reversed, but the author is currently unaware of a better procedure for doing so than simply removing unrestricted DC effects before reversal by compiling them out using C_{*3} . Thus, the reversal of a planning problem P containing unrestricted DC effects is given by $P_R = C_3(C_r(C_{*3}(P)))$. This seems inelegant, to say the least.

All in all, it appears that explicit unrestricted DC action effects are both of marginal utility and unpleasant to deal with, and should probably be avoided in problem encodings. The case against restricted DC effects is less clear, and will be considered again below.

$$\frac{f}{?f} \Rightarrow \frac{f^+}{\neg f^+ \neg f^-} \quad (6.3)$$

$$\frac{\neg f}{?f} \Rightarrow \frac{f^-}{\neg f^+ \neg f^-} \quad (6.4)$$

$$\frac{\overline{}}{?f} \Rightarrow \frac{\overline{}}{\neg f^+ \neg f^-} \quad (6.5)$$

Figure 6.2: Scheme C_4 : Extending C_3 to 4-Valued Logic

6.1.4 “Don’t Know” Effects

A “Don’t Know” or DK effect is the complement of the DC effect discussed previously. Instead of representing a non-deterministic value that satisfies both true and false preconditions on a fluent, a DK effect represents an unknown value that satisfies neither true nor false preconditions on a fluent. Thus, the only actions that can follow a DK effect on a fluent f are those that have no preconditions involving f . DK effect on f will be represented as $?f$. The extension of the C_3 compilation rules to a four-valued logic that handles DK effects $?f$ is straightforward, and is given in figure 6.2.

DK effects are especially useful as components of the initial state. A fluent f that in the initial state has both $\neg f^+$ and $\neg f^-$ represents a piece of unknown information about the initial state: a plan cannot execute actions conditional on f until it has set f either true or false (and thus set either f^+ or f^-). Note that DK values cannot usefully appear in the goal state unless DK action effects are present: there is no way for a combination of non-DK-effect operators to achieve a DK effect on a fluent.

DK action effects are also useful, in expressing effects representing loss of knowledge about a fluent’s state. Thus, if a fluent f represents whether a coin is showing heads or tails, an effect $?f$ might correspond to flipping the coin. This is a qualitative approximation to real loss of knowledge: a STRIPS planner cannot conclude, for example, that a robust plan that contains separate actions for heads and tails flips will succeed. Indeed, STRIPS is not expressive enough to describe this notion in any case. But a STRIPS planner using the DK effect representation can conclude that a plan that follows the coin flip by setting the coin to a known state will succeed. Introspectively, this seems acceptable, and it is again safe: plans containing actions with DK effects are executable.

The reversal of actions containing unrestricted DK effects, like unrestricted DC effects, is problematic. Again, compiling out the DK effects before reversal works; it is still ugly. All in all, as with DC effects, explicit DK initial conditions seem to be clearly desirable, explicit restricted DK action effects tolerable, and explicit unrestricted DK action effects problematic.

6.1.5 DC and DK Conditions

Given the treatment of explicit DC and DK operator effects, and explicit DC and DK values in the initial state, the next obvious question is the use of explicit DC and DK values in operator conditions, and in the goal state. The first question is a simple one: What meaning should be assigned to the symbols $*f$ and $?f$ in operator conditions and goal conditions?

The obvious choice is to make them explicit tests for DC and DK effects. Thus, an action precondition $*f$ could only be satisfied by an $*f$ effect, and similarly for $?f$. This can be easily implemented in the compilation, by simply compiling DC and DK conditions to the same fluent combinations as the corresponding effects. There are, however, several objections to this choice of semantics. First, this semantics is not obviously useful in encoding real-world domains. Second, the representation of explicit DK conditions induces negative preconditions in the compiled domain (although this is not a major problem). Finally, reversal becomes more complicated and difficult than ever: it appears that compiling out the conditions before reversal is the only hope of reversing a domain with these conditions.

A more attractive semantics is suggested by inspection of the reversal rules of figures 3.3 and 6.1. Note that the DC effects in rules 6.1 and 6.2 correspond to completely absent preconditions. Perhaps the right way to proceed is to assign a condition $*f$ the meaning “no condition at all on f ,” and thus substituting blank conditions in the various compilation rules with $*f$ conditions.

By making this substitution in the various compilation schemes discussed previously, such as C_2 , C_3 , and C_r , several advantages are realized.

- Notationally, things become much clearer: every fluent mentioned in an action is mentioned in both the preconditions and the effects.
- The semantics make intuitive sense: just as a DC action effect $*f$ means that subsequent actions “Don’t Care” about the value of f , a DC action condition $*f$ means that the action “Doesn’t Care” about the value of f provided by previous actions.
- The semantics of DC goal conditions are as expected: the expansion of a partial goal state to a total goal state using the expand operator of definition 3.4 does not change its correctness. (Indeed, the expansion is deleted by compilation.)

The reversal rules can finally be summarized in the natural fashion: to reverse an operator, switch its preconditions and effects, where the preconditions and effects have been normalized to the appropriate form. Without the normalization, as was noted in chapter 3 (p. 20), this technique is almost but not quite correct.

What meaning, then, should be assigned to restricted explicit DK preconditions? Arguably, the right semantics is given by compiling $?f$ in preconditions to $f^+ f^-$. This has the same attractive properties as the DC case above.

- The semantics make intuitive sense: a DK precondition $?f$ cannot be satisfied if the fluent f is known to be true (and thus $\neg f^-$) or false (and thus $\neg f^+$).
- DK conditions naturally translate nicely: a DK condition must be satisfied by a DC effect. This gives a satisfying meaning to DK goal conditions.

$$\frac{f}{f} \xLeftrightarrow{\quad} \frac{f}{f} \quad (6.6) \qquad \frac{f}{*f} \xLeftrightarrow{\quad} \frac{*f}{f} \quad (6.12)$$

$$\frac{\neg f}{\neg f} \xLeftrightarrow{\quad} \frac{\neg f}{\neg f} \quad (6.7) \qquad \frac{\neg f}{*f} \xLeftrightarrow{\quad} \frac{*f}{\neg f} \quad (6.13)$$

$$\frac{f}{\neg f} \xLeftrightarrow{\quad} \frac{\neg f}{f} \quad (6.8) \qquad \frac{?f}{f} \xLeftrightarrow{\quad} \frac{f}{?f} \quad (6.14)$$

$$\frac{\neg f}{f} \xLeftrightarrow{\quad} \frac{f}{\neg f} \quad (6.9) \qquad \frac{?f}{\neg f} \xLeftrightarrow{\quad} \frac{\neg f}{?f} \quad (6.15)$$

$$\frac{*f}{f} \xLeftrightarrow{\quad} \frac{f}{*f} \quad (6.10) \qquad \frac{f}{?f} \xLeftrightarrow{\quad} \frac{?f}{f} \quad (6.16)$$

$$\frac{*f}{\neg f} \xLeftrightarrow{\quad} \frac{\neg f}{*f} \quad (6.11) \qquad \frac{\neg f}{?f} \xLeftrightarrow{\quad} \frac{?f}{\neg f} \quad (6.17)$$

Figure 6.3: Scheme $C_{?*r}$: Full Reversal Rules

- The reversal rules involving DK effects are as expected. Reversal of actions involving restricted DK effects is by swapping the effects and preconditions. Reversal of the initial and goal conditions of a problem is by interchange.

This last item requires a proof. At this point, it is sensible to summarize the extended reversal rules, given by the compilation scheme $C_{?*r}$ of figure 6.3, and to sketch a proof of correctness for $C_{?*r}$.

Proposition 6.1 $C_{?*r}$ is correct.

PROOF:

Essentially, the structure of this proof is the same as that of the proof of proposition 3.1, except that the tables need be extended to handle the new reversal rules. Tables 6.1 and 6.2 show the legal pairs for the source and compiled problems.

Thus, the notion of restricted DC and DK effects and conditions, properly implemented, provides an expressive, easily implemented, and reversible planning language.

6.2 Composing One-Way Planning Problems

In chapter 4, a construction was given for a directional planning problem (figures 4.7 and 4.8). The purpose there was to decide the directionality of existing planners. There are other interesting

Table 6.1: Possible a_0, a_1 Pairs for $C_{?*r}$

a_0	a_1
6.6	6.6, 6.8, 6.10, 6.11, 6.12, 6.16
6.7	6.7, 6.9, 6.10, 6.11, 6.13, 6.17
6.8	6.7, 6.9, 6.10, 6.11, 6.13, 6.17
6.9	6.6, 6.8, 6.10, 6.11, 6.12, 6.16
6.10	6.6, 6.8, 6.10, 6.11, 6.12, 6.16
6.11	6.7, 6.9, 6.10, 6.11, 6.13, 6.17
6.12	6.6–6.17
6.13	6.6–6.17
6.14	6.6, 6.8, 6.10, 6.11, 6.12, 6.16
6.15	6.7, 6.9, 6.10, 6.11, 6.13, 6.17
6.16	6.10, 6.11
6.17	6.10, 6.11

Table 6.2: Possible \hat{a}_0, \hat{a}_1 Pairs for $C_{?*r}$

\hat{a}_0	\hat{a}_1
6.6	6.6, 6.9, 6.10, 6.12, 6.13, 6.14
6.7	6.7, 6.8, 6.11, 6.12, 6.13, 6.15
6.8	6.6, 6.9, 6.10, 6.12, 6.13, 6.14
6.9	6.7, 6.8, 6.11, 6.12, 6.13, 6.15
6.10	6.6–6.17
6.11	6.6–6.17
6.12	6.6, 6.9, 6.10, 6.12, 6.13, 6.14
6.13	6.7, 6.8, 6.11, 6.12, 6.13, 6.15
6.14	6.12, 6.13
6.15	6.12, 6.13
6.16	6.6, 6.9, 6.10, 6.12, 6.13, 6.14
6.17	6.7, 6.8, 6.11, 6.12, 6.13, 6.15

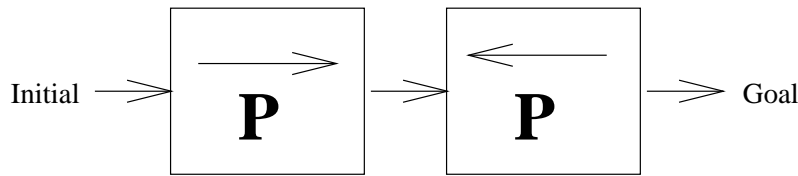


Figure 6.4: Bidirectional Planning Problem

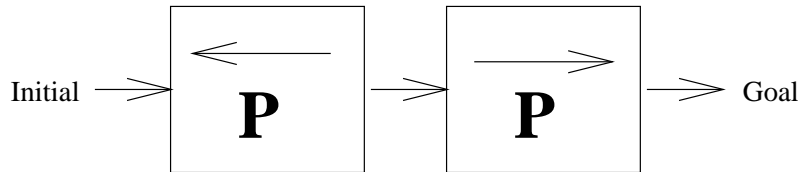


Figure 6.5: Island Planning Problem

implications of the existence of directional planning problems, as a result of the fact that one-way problems can be concatenated by concatenating their circuits.

Consider the planning problem of figure 6.4. In this diagram, the boxes with arrows represent one-way planning problems of the type described in chapter 4. The interesting thing about this problem is that neither a forward search space planner nor a backward search space planner can solve it efficiently: to do so, the planner would have to traverse half the problem in the wrong direction.

Bidirectional planning is required to solve the problem of figure 6.4. The solution, once obtained, will be a polynomial-length plan. A bidirectional planner can solve the problem in polynomial time. Together, these facts suggest that a bidirectional search space planner is strictly more powerful, in principle, than a unidirectional one.

In practice, the situation is less clear. As discussed in chapter 2, humans might always encode planning problems in a unidirectional fashion, presumably by breaking them up into unidirectional pieces. It is hard to see how this is possible in most cases, however: the author suspects that large planning problems often have a bidirectional quality. Certainly problems in many simple domains, such as the Blocks World, Rubik's Cube, or the Towers of Hanoi, are manifestly reversible, but difficult to solve. (This does not mean that bidirectional planning will help in these domains, since the problems may be intractable no matter how they are tackled. This is surely true for the optimal Blocks World, where important intractability results have been proven [25, 53].)

One can append one-way planning problem circuits in the opposite direction, as shown in figure 6.5, creating a problem that is not tractable for either a forward or a backward search space planner. Note, however, that a correct nondeterministic guess of the necessary intermediate state can break the planning problem into two halves, each of which is tractable for a directional planner.

Ginsberg [21, p. 297] refers to this intermediate state as an "island", and argues that automatic discovery of islands is the correct way to perform hierarchical planning. Indeed, the problem of figure 6.5 has been shown to be intractable only for search-space planners: it may well be that some other kind of planner could solve this problem easily (although the author would find this

surprising). One can create problems with arbitrarily many such islands: it is daunting to imagine that hard real world problems could have this sort of structure.

6.3 Tractability of Circuit-Based Planning

In chapter 4, much use was made of the fact that Nondeterministic Boolean Circuits (NBCs) have a natural representation in planning operators, and that planning problems can thus be constructed whose solution corresponds to NBC evaluation. The circuit classes considered in that chapter are all *uniform* [50]: for a given Boolean function, a circuit computing that function on n inputs can be constructed in time polynomial in n by a deterministic Turing machine.

Interestingly, the restriction to uniform circuits for planning problems appears to be arbitrary. In a *non-uniform* class of Boolean circuits, the circuit for each input size may be constructed in an arbitrary fashion. As a result, non-uniform Boolean circuits can recognize languages that are not recognizable by a Turing machine [50]. However, restricting non-uniform circuits to be of size polynomial in the size of their inputs allows them to recognize languages in the class $P/poly$. This is a larger class of languages than P (although presumably smaller than NP : if $NP \subseteq P/poly$, the polynomial hierarchy collapses to the second level [12, 53]).

A given Boolean circuit can be transformed into a planning problem in polynomial time by the construction of definition 4.16, producing a planning problem of size polynomial in the size of the circuit. Thus, it is reasonable to define classes of non-uniform planning problems, whose time to solution is polynomial only when a polynomial amount of advice.

How is all of this relevant? It places a lower bound on the complexity of PROPS planning. For a polynomially-sized domain, even the restriction that an action be used at most once leaves planning in a complexity class believed to be harder than P . This lower bound can be improved by noting that definition 4.16 and lemma 4.2 apply not just to ordinary Boolean gates, but to NBC gates. An interesting reformulation of an NBC gate is given by adding auxiliary inputs to the gate, such that the auxiliary inputs together with the normal inputs make the gate deterministic. It is thus clear that, given the correct nondeterministic guess for the auxiliary inputs of all gates in an NBC, the Boolean circuit can be evaluated in polytime. Just as non-uniform Boolean circuits can recognize languages in $P/poly$, it can be shown (though it is not proven here: see Balcázar, Díaz, and Gabarró [4, exercise 19, p.127] for an introduction) that circuits with gates of this form can recognize languages in $NP/poly$, where $P/poly \subseteq NP/poly$. Thus, the translation of NBC evaluation to planning implies that the complexity of planning can be larger than one might expect even in simple cases.

6.4 Summary: Expressiveness and Directionality of STRIPS

In this chapter, three different extensions of previous work have been considered. All three appear to point to similar conclusions. PROPS planning appears to be especially expressive. The ability to express DC and DK conditions and effects using C_4 shows that PROPS is able to express somewhat sophisticated concepts such as hidden knowledge and nondeterminism. The translation of NBC evaluation into PROPS planning also points up this expressiveness, as well as illustrating its concomitant danger: the implied intractability of even restricted forms of PROPS on general

problems. The composition of one-way functions illustrates this danger as well, by showing that simple problem structures may have profound effects on directionality.

The reversibility of $C_{? * r}$, the reverse construction of NBCPPs, and the symmetric composition of operators to produce bidirectional and island planning problems all argue that the complexity of STRIPS does not arise merely from the persistence of STRIPS fluents. Apparently, bidirectional planning is also hard. It appears that a mere choice of planning direction is not sufficient to solve the STRIPS tractability problem in general. Either a more tractable formalism or a better understanding of real world problems will be needed to make STRIPS planning tractable.

6.5 Review

At the beginning of chapter 1, some fundamental claims were made about directionality in planning.

- Successful planners must at least be capable of both forward chaining and backward chaining behavior.
- Understanding directionality issues in planning is a necessary precursor to the construction of efficient planners.

This section summarizes the material of previous chapters, highlights the underlying themes of the work, discusses future work and its impact upon the claims above, and draws some conclusions about the role of search direction in planning.

As discussed in chapter 1, there has been much speculation as to the relative merits of forward and backward search in planning. The work reported here replaces some of this speculation with concrete results. The discussion of chapter 2 examines the relationship between planning algorithms, planning problems, and STRIPS encodings. Chapters 3 and 4 put these principles into practice, by illustrating important properties of directional domains.

The construction of chapter 3 for reversing a domain is an interesting one: among other things, it shows that directionality is a property of planning problem encodings, not a property of STRIPS itself. The construction also shows that directionality is not a property arising from the shape of particular operators (since all the operators can be reversed) but from the shape of planning problems as a whole.

The construction of chapter 4 expands on this theme, by showing that particular planning problem encodings have the property that they can be tractably solved only in one direction. Interestingly, it is again global properties of the problem, rather than any special characteristics of the operators themselves, that induce this directional asymmetry. Indeed, the construction is built on the encoding of NBCs as planning problems, a construction that by its nature consists of simple operators.

Chapter 5 builds on this construction, implementing it in a general way and exploring the directionality of a variety of planners. One interesting result from these experiments is that the latest generation of planners, including Graphplan, SATplan, and `blackbox`, are largely nondirectional. The jury is still out as to whether directional planners can equal the performance of nondirectional ones on most encodings of real world problems (after all, ASP, a forward planner, performs well both on real world problems and on artificial forward one-way problems). However, these experiments provide a good example of problem domains in which directional planners exhibit vastly inferior

performance, and the experiments provide no evidence that nondirectionality hurts the performance of planners.

In chapter 6, the themes of the preceding chapters are amplified by an examination of some extensions of the work. An extension of the reversal schema of chapter 3 exhibits a proper superset of standard PROPS that is both expressive and reversible. A discussion of the composition of the one-way problems of chapter 4 highlights the ways in which global properties of problems contribute to interesting constraints on search direction. Finally, a discussion of the computation complexity of NBC planning points up the expressiveness and consequent potential intractability of even simple PROPS problems.

6.6 Future Work

Several immediate extensions of this work seem promising.

- The PROPS planning language of Chapters 3 and 6 should be implemented as a preprocessor for existing planners, and its expressiveness and tractability evaluated against encodings of real world problems.
- As planners become more powerful, the hash function \mathcal{H}_n of chapter 5 should be replaced in the detector by a one-way function considered secure by the general cryptographic community. Recent work in cryptography for “smart cards” and similar tiny computing environments should be helpful here, by providing strong one-way functions that are evaluable with minimal computational resources.
- The results of Chapters 3–6 should be applied to the construction of planning algorithms using high-speed bidirectional or nondirectional search. It is likely that nondirectional state space search, if properly conceived and implemented, can be the basis of an efficient planning algorithm.

The results of this work also suggest that more attention should be paid to the construction of planning domain description languages. What is needed are languages that are more tractable than STRIPS, but still expressive enough to describe real-world problems. While the intractability of STRIPS has long been understood, it nonetheless seems that much of the work on planning has been designed to increase its expressiveness, usually at the expense of tractability. The constructions of previous chapters indicate that this sacrifice may not be necessary: many interesting real-world concepts can be expressed indirectly using the simple PROPS formalism.

Resource-bounded scheduling and enterprise planning are less expressive formalisms that appear to be more tractable than STRIPS, while still being able to encode large classes of real planning problems. It is unclear whether formalisms intermediate in expressive power and tractability between scheduling and STRIPS can be developed, but it definitely seems worth further exploration. Given such a formalism, the compilation of STRIPS to it would be one of the first challenges. This work suggests that the desired formalism need not have an implicit notion of persistence: this directionality result may be useful in its development.

6.7 Conclusions

A poor choice of direction can doom a STRIPS planner on particular problems; nonetheless, neither bidirectionality nor the ability to choose an appropriate direction makes planning easy. Rather than argue the advantages of backward planning (as the advocates of POCL planning have) or forward planning (as the advocates of languages even more expressive and potentially intractable than STRIPS have), this work provides a strong argument that the concentration of effort should be on finding planning languages and algorithms that are bidirectional, and which make real world problems tractable.

Bibliography

- [1] James Allen, James Hendler, and Austin Tate, editors. *Readings in Planning*. Morgan Kaufmann, San Mateo, CA, 1990.
- [2] Anbulagan and Chu Min Li. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 366–371, August 1997.
- [3] F. Bacchus and Y. W. Teh. Making forward chaining relevant. In Simmons et al. [54], pages 54–61.
- [4] José Luis Balcázar, Josep Díaz, and Joaquim Gabarró. *Structural Complexity I*. Springer-Verlag, 1988.
- [5] Anthony Barrett and Daniel S. Weld. Partial-Order planning: Evaluating possible efficiency gains. Technical Report TR 92-05-01, University of Washington, May 1992.
- [6] Roberto J. Bayardo Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence* [47], pages 203–208.
- [7] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways For Your Mathematical Plays*, volume 2. Academic Press, 1982.
- [8] A. Blum and M. Furst. *README for Graphplan*. Carnegie Mellon University, 1996. <<http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/usr/avrim/Planning/Graphplan/README>>.
- [9] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.
- [10] Alexander Bogomolny. Reverse solitaire. Web document, 1996. <<http://www.cut-the-knot.com/proofs/rsolitaire.html>>.
- [11] Blai Bonet, Gábor Loerincs, and Héctor Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the National Conference on Artificial Intelligence* [47], pages 714–719.
- [12] R.B. Boppana and M. Sipser. The complexity of finite functions. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A. MIT Press/Elsevier, 1990.

- [13] Britannica Online. Animal Learning: Types of learning: Complex Problem Solving: Insight and reasoning. Web document, October 1998. <<http://www.eb.com:180/cgi-bin/g?DocF=macro/5003/68/21.html>>.
- [14] Tom Bylander. Complexity results for planning. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, volume 1, pages 274–279, August 1991.
- [15] Anthony R. Cassandra. Optimal policies for Partially Observable Markov Decision Processes. Technical Report CS-94-14, Brown University, August 1994.
- [16] Ken Currie and Austin Tate. O-Plan—control in the open planning architecture. In *Proceedings of the Conference On Expert Systems 1985*, pages 225–240, London, England, 1985. Cambridge University Press. Also in Readings in Planning [1].
- [17] Richard Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971. Also in Readings in Planning [1].
- [18] Eugene Fink and Jim Blythe. A complete bidirectional planner. In Simmons et al. [54], pages 78–85.
- [19] B. Cenk Gazen and Craig A. Knoblock. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In *Recent Advances In AI Planning: 4th European Conference*, September 1997.
- [20] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, Los Altos, CA, 1987.
- [21] Matt Ginsberg. *Essentials of Artificial Intelligence*. Morgan Kaufman, San Mateo, CA, 1993.
- [22] Matthew L. Ginsberg. Approximate planning. *Artificial Intelligence*, 76(1-2):89–123, 1995.
- [23] Matthew L. Ginsberg and D.E. Smith. Reasoning about action I: A possible-worlds approach. *Artificial Intelligence*, 35:311–342, 1988.
- [24] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the First International Joint Conference on Artificial Intelligence*, pages 741–747, San Mateo, CA, 1969. Morgan Kaufmann. Also in Readings in Planning [1].
- [25] Naresh Gupta and Dana S. Nau. Complexity results for blocks-world planning. In *Proceedings of the National Conference on Artificial Intelligence* [44], pages 629–633.
- [26] D. Joslin and M. E. Pollack. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proceedings of the National Conference on Artificial Intelligence* [45], pages 1004–1009.
- [27] Subbarao Kambhampati. Refinement planning as a unifying framework for plan synthesis. *AI Magazine*, 18(2):67–97, Summer 1997.

- [28] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence* [46], pages 1194–1201.
- [29] Henry Kautz and Bart Selman. BLACKBOX: A new approach to the application of theorem-proving to problem solving. In *Working Notes of the Workshop on Planning as Combinatorial Search (held in conjunction with AIPS-98)*, Pittsburgh, PA, 1998.
- [30] B. W. Kernighan and D. M. Ritchie. The M4 macro processor. In *Unix Programmer's Manual*. Bell Laboratories, Murray Hill, NJ, 1979.
- [31] Hector J. Levesque. What is planning in the presence of sensing? In *Proceedings of the National Conference on Artificial Intelligence* [46], pages 1139–1146.
- [32] Z. Manna and R. Waldinger. A theory of plans. In M. P. Georgeff and A. L. Lansky, editors, *Reasoning About Actions & Plans — Proceedings of the 1986 Workshop*, pages 11–46. Morgan Kaufmann Publishers: San Mateo, CA, 1986.
- [33] Bart Massey. Planner directionality testbed. FTP-able compressed TAR archive, 1999. <ftp://ftp.cirl.uoregon.edu/pub/users/bart/testbed.tgz>.
- [34] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the National Conference on Artificial Intelligence* [44], pages 634–639.
- [35] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.
- [36] Drew McDermott. The current state of AI planning research. In *International Conference on Industrial and Engineering Applications of AI and Expert Systems*, 1994.
- [37] Drew McDermott et al. The Planning Domain Definition Language manual. Technical Report Computer Science 1168 (CVC Report 98-003), Yale University, 1998. <ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz>.
- [38] Alan Newell and Herbert Simon. GPS, a program that simulates human thought. In *Computers and Thought*. New York: McGraw-Hill, 1963. Also in *Readings in Planning* [1].
- [39] E. P. D. Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4(4):356–372, November 1988.
- [40] J.S. Penberthy. *Planning With Continuous Change*. PhD thesis, University of Washington, 1993. Also available as technical report UW-CSE-93-12-01.
- [41] J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial-order planner for ADL. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, October 1992.
- [42] John R. Pierce. *An Introduction to Information Theory: Symbols, Signals, and Noise*. Dover, 1980.

- [43] Martha E. Pollack, David Joslin, and Massimo Paolucci. Flaw selection strategies for partial-order planning. *Journal of Artificial Intelligence Research*, 6:223–262, 1997.
- [44] *Proceedings of the Ninth National Conference on Artificial Intelligence*, Anaheim, California, July 1991. AAAI Press/Morgan Kaufman.
- [45] *Proceedings of the Twelfth National Conference on Artificial Intelligence*. AAAI Press/MIT Press, August 1994.
- [46] *Proceedings of the Thirteenth National Conference on Artificial Intelligence*. AAAI Press/MIT Press, August 1996.
- [47] *Proceedings of the Fourteenth National Conference on Artificial Intelligence*. AAAI Press/MIT Press, July 1997.
- [48] Jussi Rintanen. A planning algorithm not based on directional search. In *Proceedings of the Sixth International Conference on Knowledge Representation and Reasoning*, pages 617–624, June 1998.
- [49] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974. Also in Readings in Planning [1].
- [50] John E. Savage. *Models of Computation*. Addison-Wesley, 1987.
- [51] Bruce Schneier. *Applied Cryptography*. Wiley, second edition, 1995.
- [52] Lenhart Schubert and Alfonso Gerevini. Accelerating partial order planners by improving plan and goal choices. Technical Report TR 607, University of Rochester, Computer Science Department, January 1996.
- [53] Bart Selman. Near-optimal plans, tractability, and reactivity. In *Proceedings of the Fourth International Conference on Knowledge Representation and Reasoning*, pages 521–529, 1994.
- [54] Reid Simmons, Manuela Veloso, and Steven Smith, editors. *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*. AAAI Press, 1998.
- [55] Douglas R. Stinson. *Cryptography: Theory and Practice*. CRC Press, 1995.
- [56] Gerald Jay Sussman. The virtuous nature of bugs. In Allen et al. [1].
- [57] Paul Tipler. *Physics*. Worth Publishers, Inc., New York, NY, second edition, 1982.
- [58] Dan Weld and Oren Etzioni. The first law of robotics (a call to arms). In *Proceedings of the National Conference on Artificial Intelligence* [45], pages 1042–1047.
- [59] David Wilkins. *Practical Planning: Extending The Classical AI Planning Paradigm*. Morgan Kaufman, San Mateo, CA, 1988.