Efficient BDD-Based Planning for Non-Deterministic, Fault-Tolerant, and Adversarial Domains

Rune Møller Jensen

June 2003

CMU-CS-03-139

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:

Manuela M. Veloso (Co-Chair), Carnegie Mellon University
Randal E. Bryant (Co-Chair), Carnegie Mellon University
Reid Simmons, Carnegie Mellon University
Paolo Traverso, IRST, Trento, Italy

This research was supported in part by the Danish Ministry of Science, Technology and Innovation and the United States Air Force under Grants Nos F30602-00-2-0549 and F30602-98-2-0135.

Keywords: Automated Planning, Heuristic Search, Binary Decision Diagrams, Artificial Intelligence, Symbolic Model Checking, Controller Synthesis.

To my grandparents

Abstract

Automated planning considers selecting and sequencing actions in order to change the state of a discrete system from some initial state to some goal state. This problem is fundamental in a wide range of industrial and academic fields including robotics, automation, embedded systems, and operational research. Planning with non-deterministic actions can be used to model dynamic environments and alternative action behavior. One of the currently best known approaches is to employ reduced ordered Binary Decision Diagrams (BDDs) to represent and generate plans using techniques developed in symbolic model checking. However, the approach is challenged by a frequent blow-up of the BDDs representing the search frontier and a limited number of solution classes.

This thesis addresses both of these problems. With respect to the first, it contributes a general framework called *state-set branching* that seamlessly combines classical heuristic search and BDD-based search. Our experimental results show that the performance of state-set branching often dominates both blind BDD-based search and ordinary heuristic search. In addition, it consistently outperforms any previous approach, we are aware of, to guide a BDD-based search. We show that state-set branching naturally generalizes to non-deterministic planning and introduce heuristically guided versions of the current BDD-based non-deterministic planning algorithms.

With respect to the second problem, the thesis introduces two frameworks called *fault tolerant planning* and *adversarial planning*. Fault tolerant planning addresses domains where non-determinism is caused by rare errors. The current solution classes handle this situation poorly by taking all fault combinations into account or produce too weak solutions. The thesis contributes a new class of solutions called *fault tolerant plans* that are robust to a limited number of faults. In addition, it introduces specialized BDD-based algorithms for synthesizing fault tolerant plans.

Adversarial planning considers situations where non-determinism is caused by uncontrollable, but known, environment actions. The current solution classes of BDD-based non-deterministic planning assume a "friendly" environment and may never reach a goal state if the environment is hostile and informed. The thesis contributes efficient BDD-based algorithms for synthesizing winning strategies for such problems.

Acknowledgments

This thesis has been a great journey in science, culture and life. Manuela, I would like to thank you for the great guidance and support you have given me, not only on research issues but on all aspects of this complicated voyage. Without your help and encouragement, I would not have made it this far.

Randy, thank you very much for your extensive support and for taking care of me while Manuela was on leave at MIT. I have always looked forward to discuss ideas with you. Your deep insight in my work has helped me to cut wild paths early and encouraged me to pursue fruitful ideas.

I would also like to thank the rest of my thesis committee, Reid and Paolo. Thank you Reid for your detailed feedback on the thesis draft and for providing the Deep Space One domain. Thank you Paolo for being able to attend my thesis defense and for your key feedback on the thesis draft.

I also wish to thank Henrik Reif Andersen for hosting me as a visitor at the IT University of Copenhagen in the summer 2000 and 2001 and for taking the job as Danish contact person for my scholarship from the Danish Ministry of Science, Technology and Innovation.

Several people, in addition to Reid, have contributed key benchmark problems to this thesis. I would like to thank Sylvie Thiebaux and Piergiorgio Bertoli for providing the PSR domain, Anders P. Ravn for giving me pointers to the SIDMAR case study, and Kolja Sulimma for providing the channel routing benchmark problems.

At an early point, I became interested in the application of non-deterministic planning for controller synthesis. I wish to thank Bruce Krogh for our many exciting meetings on this topic and for giving me pointers to relevant work in control theory. I also wish to thank Gregg Ekberg at Highline Controls and Nicola Muscettola at NASA Ames for showing interest in this particular application of my work.

Finally, I wish to thank my family in Denmark and Alexander Gray and Kent H. Andersen for their great friendship and support during my time at Carnegie Mellon.

Notation

Symbol	**Interpretation**
$a \in A$	the element a is a member of the set A
$a \notin A$	the element a is not a member of the set A
$A \subseteq B$	the set A is a subset of the set B
$A \subset B$	the set A is a proper subset of the set B
$A \not\subseteq B$	the set A is not a subset of the set B
$A \supseteq B$	the set A is a superset of the set B
$\emptyset$	the empty set
$\mathcal{U}$	the universal set
$\|A\|$	the cardinality of A
$\overline{A}$	the complement of the set A
$A \setminus B$	the difference of the sets A and B
$A \cap B$	the intersection of the sets A and B
$A \cup B$	the union of the sets A and B
$\bigcap_{i=1}^{n} A_i$	the intersection of the sets $A_1, \cdots, A_n$
$\bigcup_{i=1}^{n} A_i$	the union of the sets $A_1, \cdots, A_n$
$\bigcap_{i \in I}^{n} A_i$	the intersection of the family of sets $\{A_i \mid i \in I\}$
$\bigcup_{i \in I}^{n} A_i$	the union of the family of sets $\{A_i \mid i \in I\}$
2^A	the power set of the set A
$A \times B$	the Cartesian product of the set A and B
$\neg p$	negation of the proposition p
$p \wedge q$	conjunction of the proposition p and q
$p \vee q$	inclusive disjunction of the proposition p and q
$true$	tautology
$false$	contradiction
$\forall$	the universal quantifier
$\exists$	the existential quantifier
$p \Rightarrow q$	the proposition p implies the proposition q
$p \Leftrightarrow q$	$p \Rightarrow q$ and $q \Rightarrow p$
$\bigwedge_{i=1}^{n} p_i$	the conjunction of the propositions $p_1, \cdots, p_n$
$\bigvee_{i=1}^{n} p_i$	the disjunction of the propositions $p_1, \cdots, p_n$

Symbol	**Interpretation**
$\bigwedge_{i \in I}^{n} p_i$	the conjunction of the family of propositions $\{p_i \mid i \in I\}$
$\bigvee_{i \in I}^{n} p_i$	the disjunction of the family of propositions $\{p_i \mid i \in I\}$
$\mathbb{B}$	the Boolean constants $\{true, false\}$
$\mathbb{N}$	the set of natural numbers
$\mathbb{R}$	the set of real numbers
$\mathbb{N}^+$	the set of positive natural numbers
$\mathbb{R}^+$	the set of positive real numbers
$\mathbb{R}_0^+$	the set of non-negative real numbers
$\mathbb{K}$	the set of Kripke structures
$f\|_{v \leftarrow \alpha}$	the Boolean formula f restricted to the value $\alpha \in \mathbb{B}$ of the Boolean variable v
$f[x/y]$	the substitution of variable x with variable y in expression f
$\{s : p(s)\}$	the set of elements s satisfying proposition $p(s)$
$\vec{x}$	a vector $(x_1, \cdots, x_n)$
$\equiv$	defining equality
$\leftarrow$	assignment in algorithm description
$\mathbf{M}$	map in algorithm description
$\|\mathbf{M}\|$	the number of entries in the map $\mathbf{M}$ in algorithm description
$\mathbf{M}[k]$	the entry with key k in map $\mathbf{M}$ in algorithm description (k_i denotes the ith key of $\mathbf{M}$)
M	the union of the entries in map $\mathbf{M}$ in algorithm description

Contents

Chapter 1

Introduction

Planning is a fundamental aspect of human activity. It considers how to select and sequence actions in order to achieve specific goals. This problem arises in a wide range of situations. For instance, we need to select and sequence rotations very carefully in order to change Rubik's Cube from some initial configuration to its goal configuration where each side has identically colored tiles. Planning is also needed to control a power plant in order to recover from any possible failure of the plant. This problem is fairly different from solving Rubik's Cube since the control actions depend on an interacting environment. In general, we can divide planning problems into two main categories: *deterministic planning problems* and *non-deterministic planning problems*. Rubik's Cube is a good example of a deterministic planning problem. Each action has only one possible outcome. In other words, actions are deterministic. This is not the case for non-deterministic planning problems. Consider the power plant problem. Due to failures, actions may either succeed or fail. Thus, several outcomes of actions are possible. In general, dynamic environments cause actions to be non-deterministic since we are unable to determine their exact effect.

Planning problems are often very hard to solve in practice. There are several reasons for this. First, real-world domains tend to be extremely large. Assume for instance that the power plant described above consists of n units that each can be in at least two different states. The total number of states of the power plant is then at least 2^n. Thus, the state space of the power plant grows exponentially with the number of units. This is a common problem for real-world domains and has been termed the *state space explosion problem*. Second, plans for real-world problems are often long. A plan for controlling a power plant or loading a container ship may involve sequencing thousands of actions. Third, the combinatorial complexity of planning may be high. The Rubik's Cube is a hard puzzle because it is impossible to move one tile without affecting the position of several other tiles. Similarly, routing wires between units on an integrated circuit is complicated because routing one wire strongly constrains how the remaining wires can be routed. Finally, fourth, a planning problem may not only be to generate a *valid* plan but an *optimal* one. We may want to minimize the energy consumption or time used to load a container ship or we may want to use a minimum number of connections between layers (vias) when routing the wires of an integrated circuit. Such optimal plans can be much harder to find than just valid ones.

Automated planning is a subfield of Artificial Intelligence (AI) concerned with how to generate plans automatically. The traditional approach is to make a discrete abstraction of the real-world domain and search for a solution. More formally, a *planning domain* consists of a finite set of states, a finite set of actions, and a transition relation defining the effect of actions. A *planning problem* consists of a planning domain, an initial state,[1] and a set of goal states. For a deterministic planning problem, a *plan* is a sequence of actions forming a path leading from the initial state to one of the goal states. For a non-deterministic planning problem where actions may lead to several possible next states, a plan may be defined as a function associating states with relevant actions to apply at the state in order to reach a goal state. Planning domains and planning problems are traditionally described in a *planning language* that uses propositional or first order logic to define actions and states. This encoding scheme causes the formal complexity of planning to be PSPACE-complete [29].

The primary challenge of automated planning is to provide efficient algorithms and data structures to represent and synthesize plans that scale to real-world problems. During its more than 40 years of existence, [2] the field has contributed a vast number of effective search techniques including means-end analysis [126], hierarchical abstraction [148], partial-order planning [149], case-based planning [72, 164], graph-planning [18], heuristic search [75, 20], and planning and learning [163]. In addition, there has recently been successful work on reducing planning to satisfiability [99], model checking [33], and integer programming [19]. In recent years, the efficiency of planning systems has grown considerably. However, scaling to moderately large real-world problems is still an open problem.

The secondary challenge of automated planning is to provide planning algorithms that can handle essential properties of real-world domains such as time, dynamic environments, partial observability of states, and concurrency. The challenge is not simply to extend the expressiveness of planning languages and generalize algorithms to manage all these properties, but instead to carefully develop new representations and algorithms with an attractive trade-off between expressiveness and scalability. Non-deterministic planning is an example of such a positive trade-off. Non-determinism can model essential properties of dynamic

[1]If the initial state is uncertain, we may represent it by a set of states.

[2]The General Problem Solver (GPS) [126] is widely considered the first automated problem solver.

domains such as uncontrollable actions and alternative action behavior without introducing computational expensive elements in the model like continuous time and probability distributions.

Recently, efficient algorithms using reduced ordered Binary Decision Diagrams (BDDs) [26] to represent plans and applying implicit search techniques developed for symbolic model checking [119] have been shown to outperform a wide range of the previous approaches to non-deterministic planning [34]. However, a major challenge of this line of research is that non-deterministic domain models often are coarse abstractions that make it hard to define strong solution models.

To summarize, planning is about selecting and sequencing actions in order to obtain specific goals. In general, planning problems can be divided according to the environment which either can be non-interacting or interacting. In both cases, most real-world planning problems are very hard and the primary goal of automated planning is to provide efficient algorithms and data structures that scale to real-world applications. A secondary goal is to develop planning systems that handle essential properties of real-world domains such as dynamic environments and time. An interesting recent development in this direction is BDD-based non-deterministic planning. However, the range of solution classes developed within this framework is still limited.

1.1 Approach

The overall objective of the thesis is to contribute efficient algorithms for non-deterministic planning. We consider the *universal planning* [154] approach to non-deterministic planning. In universal planning, actions are assumed to be non-deterministic in the sense that they may have several possible outcomes. States, on the other hand, are assumed to be fully observable. A universal plan is a function mapping states to sets of relevant actions to apply in order to reach a goal state. It is executed by iteratively observing the current state and applying one of the actions in the plan associated with that state. From a control theoretic point of view, universal planning corresponds to *automated controller synthesis* of discrete, untimed, and memory less controllers.

A main assumption of the thesis is that the computational advantages of non-deterministic domain models outweigh the problems of defining practically useful solution classes due to their limited expressive power. We believe that the absence of continuous elements, like time and probability distributions may be essential for developing algorithms that scale to real-world problems. In addition, we trust that the limitation of the expressive power of non-deterministic abstractions can be overcome by adding further information to the model identifying different sources of non-determinism (e.g., by distinguishing between successful and failure outcomes of actions).

The thesis relies on the efficiency of BDDs to represent and generate non-deterministic plans. A BDD is a rooted Directed Acyclic Graph (DAG) representing a Boolean function. Its main advantage is that the number of nodes in the BDD graph often is much smaller than the number of truth assignments of the Boolean function it represents. State-of-the-art BDD-based non-deterministic planning algorithms iteratively construct a BDD representing the plan. This is done by an implicit breadth-first backward search from the goal states to the initial state carried out entirely with BDDs. Due to the compactness of BDDs, this approach may reduce both the time and space complexity exponentially compared to explicit search techniques.

The current approaches to BDD-based non-deterministic planning face two major challenges. The first is that BDD-based non-deterministic planning despite, its unprecedented efficiency, still does not scale to large real-world problems. Often the BDDs representing the backward breadth-first search frontier blow up [86]. This tendency seems to be worse for typical planning problems compared to typical model checking problems. One reason for this might be that planning domains often represent hard combinatorial problems such as channel routing in VLSI design, whereas model checking benchmarks often are industrial cases with no particular intention of being combinatorially hard. Another difference is the diameter of the finite transition graphs representing the planning domain. Planning problems are deliberately designed to have transition graphs with large diameters causing plan solutions to be long. Again, model checking benchmarks are not particularly chosen to fulfill this requirement.

The second challenge of BDD-based non-deterministic planning is that the limited expressive power of the non-deterministic abstraction makes it hard to define solution classes that are useful in practice. Non-deterministic domain models often hide too much information about the source of non-determinism to allow useful solutions models to be defined.

Given these challenges, the goal of the thesis is to answer two questions

1. *Can the computational efficiency of BDD-based non-deterministic planning be improved ?*

2. *Is it possible to improve the current solution classes for BDD-based non-deterministic planning ?*

The thesis addresses the first question by introducing a seamless combination of BDD-based search and *heuristic search*. [3] The advantage of heuristic search algorithms such as

[3] We will use the terms heuristic search, guided search, directed search, and informed search interchangeably.

pure heuristic search and A* [75] compared to *uninformed* or *blind* search algorithms such as depth-first search and breadth-first search, is that they use heuristics to prioritize the node expansion in the search tree and in this way *guide* the search toward a solution. In most cases, the number of states of a guided search frontier grows slower with the search depth than the number of states of an unguided search frontier. We therefore expect that the size of BDDs representing a guided search frontier grow slower than the size of BDDs representing a blind search frontier. Several attempts have been made to implement BDD-based versions of these algorithms. They have, however, either been inefficient [53, 74, 180] or too narrow in scope [179]. The approach introduced in the thesis is called *state-set branching*. The philosophy of state-set branching is that the information represented by BDDs must be semantically closely related in order for the BDD operations to work efficiently. In contrast to previous work, state-set branching avoids arithmetic computations at the BDD level in each iteration of the search algorithm. Instead, these computations are integrated in the BDD operation computing the search frontier. State-set branching is general. It applies to any heuristic function, any evaluation function, and any transition cost function. In addition, state-set branching extends beyond classical heuristic search and deterministic planning. In its non-deterministic version called *non-deterministic state-set branching*, it can be used to dramatically improve the performance of non-deterministic BDD-based planning algorithms not only in terms of computational efficiency but also in terms of the size of the produced plans.

The thesis addresses the second thesis question by introducing two extensions of the ordinary non-deterministic domain model. The first extension is based on the key observation that non-determinism in real-world domains often is caused by infrequent errors that make otherwise deterministic actions fail. In many cases, no actions can be guaranteed to succeed. For such problems plans taking all combinations of faults into account seldom exists. The approach introduced in the thesis is called *fault tolerant planning*. Fault tolerant plans are robust to a limited number of faults happening during execution.

The second extension considers situations where the main source of non-determinism is uncontrollable actions selected by a possibly hostile environment. By extending the ordinary non-deterministic domain model to explicitly represent environment actions, it is possible to reason about the actions of the environment during planning. The approach introduced in the thesis is called *adversarial planning*. The key idea is to prune unfair states from the plans where the environment has an action for which no counter action exists that may cause progress toward the goal states.

1.2 Thesis Contributions

The thesis has five major contributions:

1. **State-Set Branching**
   State-set branching appears to be the currently most general and most computationally efficient framework for combining classical heuristic search and BDD-based search. It applies to any best-first search algorithm, any heuristic function, any evaluation function, and any transition cost function. A state-set branching implementation of A* often dominates both the ordinary explicit-state implementation of A* and blind BDD-based search.

2. **Non-Deterministic State-Set Branching**
   Non-deterministic state-set branching is, as far as we know, the first framework for guiding BDD-based non-deterministic planning algorithms.[4] Even for fairly weak heuristics, extensive performance improvements over the current non-deterministic BDD-based planning algorithms can be obtained not only in terms of computation speed but also in terms of the size of the produced plans.

3. **Fault Tolerant Planning**
   To our knowledge, the fault tolerant planning algorithms introduced in the thesis are the first algorithms to synthesize fault tolerant control strategies given a domain description that explicitly represents successful and failure effects of actions.

4. **Adversarial Planning**
   Adversarial planning is, as far as we know, the first work that studies fully implemented and complete symbolic algorithms for synthesizing strategies for winning concurrent reachability games with probability 1 or positive probability. To our knowledge, it also is the first work that provides such algorithms in a format that enables guided search techniques to be applied.

5. **NADL$^+$**
   NADL$^+$ is an extension of NADL [93]. To our knowledge, it is the first representation language suitable for planning that both explicitly represents uncontrollable environment actions and failure effects of actions.

As described in the previous section, these contributions are along two orthogonal axises: *computational efficiency* and *solution quality*. State-set branching falls on the first

[4] By non-deterministic planning, we refer to the definition given in Section 3.2.

axis, Fault tolerant planning and adversarial planning mainly fall on the second. In addition to this work, the thesis contributes formal correctness and optimality proofs for algorithms where these properties are nontrivial. Moreover, the thesis provides the *Bdd-based InFoRmed planning and cOntroller Synthesis Tool* (BIFROST) for solving search and planning problems described in PDDL [118] and NADL$^+$. BIFROST is fully implemented and currently includes 8 deterministic and 10 non-deterministic planning and search algorithms.

1.3 Document Outline

The remainder of the thesis is organized as follows. Chapter 2 presents background material including BDDs, symbolic model checking, and heuristic search. Chapter 3 presents basic techniques for encoding deterministic STRIPS [58] and non-deterministic NADL and NADL$^+$ planning problems with BDDs and presents unguided BDD-based search algorithms for synthesizing deterministic and non-deterministic plans. Chapter 4 introduces state-set branching. It is shown how the framework can be used to implement algorithms for classical heuristic search and deterministic planning. Chapter 5 introduces non-deterministic state-set branching and describes guided versions of the blind BDD-based non-deterministic planning algorithms. Chapter 6 defines fault tolerant planning and introduces both blind and guided BDD-based algorithms for generating fault tolerant plans. Chapter 7 presents adversarial planning and describes two BDD-based adversarial planning algorithms. Finally, Chapter 8 discusses related work, and Chapter 9 presents conclusions and future work. BIFROST is described in Appendix A and correctness and optimality proofs are given in Appendix B.

Chapter 2

Background

This chapter presents a range of formalisms and logics used in the thesis and gives an introduction to symbolic model checking and classical heuristic search. Section 2.1 describes Quantified Boolean Formulas (QBF), Kripke structures, and Computation Tree Logic (CTL). Section 2.2 presents the BDD data structure and modern BDD packages. Section 2.3 describes symbolic model checking. It shows how BDDs can be used to represent and search a finite transition system and describes a technique known as transition relation partitioning used to lower the complexity of BDD-based search. Finally, Section 2.4 gives a brief introduction to classical heuristic search.

2.1 Logics and Formalisms

This section presents Quantified Boolean Formulas (QBF) [1], Kripke structures [111], and the Computation Tree Logic (CTL) [10, 55]. QBF provides a concise notation for complex operations on Boolean formulas which we will use extensively to define BDD operations. Kripke structures and CTL are basic tools for specifying behavior of non-deterministic systems [40, 42]. We will use them to define various classes of non-determini-stic plans.

2.1.1 Quantified Boolean Formulas

Quantified Boolean Formulas (QBF) is ordinary propositional logic extended with quantification of Boolean variables.

Definition 2.1 (QBF syntax) *Given a set $V = \{v_1, \cdots, v_n\}$ of propositional variables, QBF(V) formulas are inductively defined by*

- *every variable in V is a formula,*
- *if f and g are formulas, then so are $\neg f$, $f \wedge g$, and $f \vee g$, and*
- *if f is a formula and $v \in V$, then $\exists v \,.\, f$ and $\forall v \,.\, f$ are formulas.*

A *truth assignment* for QBF(V) is a function $\sigma : V \rightarrow \mathbb{B}$. We will use the notation $\sigma\langle v \leftarrow a\rangle$ for the truth assignment defined by

$$\sigma\langle v \leftarrow a\rangle(w) = \begin{cases} a & : \text{ if } v = w \\ \sigma(w) & : \text{ otherwise.} \end{cases}$$

Definition 2.2 (QBF Semantics) *If f is a formula in QBF(V) and σ is a truth assignment, we will write $\sigma \models f$ to denote that f is true under the assignment σ. The relation $\models$ is defined inductively in the obvious manner*

- $\sigma \models v$ *iff* $\sigma(v) = \mathit{true}$,
- $\sigma \models \neg f$ *iff* $\sigma \not\models f$,
- $\sigma \models f \vee g$ *iff* $\sigma \models f$ *or* $\sigma \models g$,
- $\sigma \models f \wedge g$ *iff* $\sigma \models f$ *and* $\sigma \models g$,
- $\sigma \models \exists v \,.\, f$ *iff* $\sigma\langle v \leftarrow \mathit{false}\rangle \models f$ *or* $\sigma\langle v \leftarrow \mathit{true}\rangle \models f$,
- $\sigma \models \forall v \,.\, f$ *iff* $\sigma\langle v \leftarrow \mathit{false}\rangle \models f$ *and* $\sigma\langle v \leftarrow \mathit{true}\rangle \models f$.

For a vector $\vec{v} = (v_1, \cdots, v_m)$ of propositional variables in V, we define the abbreviations

$$\exists \vec{v} \,.\, f \equiv \exists v_1 \,.\, (\cdots (\exists v_m \,.\, f) \cdots) \tag{2.1}$$

$$\forall \vec{v} \,.\, f \equiv \forall v_1 \,.\, (\cdots (\forall v_m \,.\, f) \cdots). \tag{2.2}$$

The *support* of a formula f is the set of variables that f depends on $\{v \in V \,:\, f|_{v \leftarrow \mathit{true}} \neq f|_{v \leftarrow \mathit{false}}\}$.

2.1.2 Kripke Structures

A Kripke structure [111, 40] is finite state transition graph that can be used to capture the intuition about the behavior of a finite transition system. The standard definition of a Kripke structure is a set of states, a set of transitions between states, and a function that labels each state with a set of propositions that are true in this state. For the purpose of this thesis, however, we consider a simplified version of the standard definition without propositions.[1]

[1] Similar restrictions have been used in model checking [39].

Definition 2.3 (Kripke Structure) *A Kripke structure $\mathcal{K}$ is a pair $\mathcal{K} = \langle S, R\rangle$ where S is a finite set of states and $R \subseteq S \times S$ is a total transition relation.*[2]

A state and a transition of a Kripke structure denote a state and a possible state change of the finite transition system the Kripke structure represents. A path in a Kripke structure represents an *execution* of the system. A *path* π in $\mathcal{K}$ is an infinite sequence $q_0q_1\cdots$ of states in S such that, for $i > 0$, $\langle s_i, s_{i+1}\rangle \in R$.

Example 2.1 A Kripke structure with four states $S = \{A, B, C, D\}$ and five transitions $R = \{\langle A, B\rangle, \langle B, D\rangle, \langle C, A\rangle, \langle C, D\rangle, \langle D, C\rangle\}$ is shown in Figure 2.1. $\Diamond$

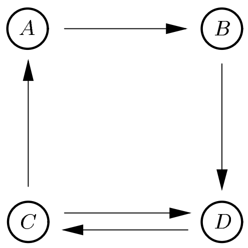


Figure 2.1: A Kripke structure with four states and five transitions.

2.1.3 Computation Tree Logic

Computation Tree Logic (CTL) [10, 55] is a branching-time temporal logic to specify the behavior of a system represented by a Kripke structure. In branching-time temporal logics, the underlying structure of time is assumed to have a branching tree-like nature where each moment may have many successor moments. For a Kripke structure, this *execution tree* is formed by designating a state in the Kripke structure as an initial state and then unwinding the structure into an infinite tree with the designated state as root. Each path in the tree is a path in the Kripke structure and represents a possible execution of the system the Kripke structure models.

Example 2.2 The execution tree starting in state C of the Kripke structure shown in Figure 2.1 is illustrated in Figure 2.2. $\Diamond$

We consider a small subset of CTL formulas sufficient for our purposes. CTL formulas are composed of *path quantifiers* and *temporal operators*. The path quantifiers are used to describe the branching structure in the execution tree. There are two such quantifiers `A`

[2]A transition relation is total iff $\forall s\,.\,\exists s'\,.\,\langle s, s'\rangle \in R$.

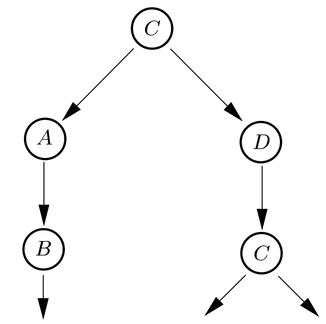


Figure 2.2: The execution tree produced from state C of the Kripke structure shown in Figure 2.1.

("for all execution paths") and E ("for some execution path"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths starting at that state have some property. The temporal operators describe properties of a path through the tree. We consider one of these U ("until"). It is used to combine two properties ϕ and ψ. It holds if there is a state on the path where ψ holds, and at every preceding state on the path, ϕ holds.

Definition 2.4 (CTL Syntax) *Given a finite set of states S, the syntax of CTL formulas are inductively defined as follows*

- *Each element of 2^S is a formula,*
- $\neg\psi$, $\mathtt{E}(\phi\,\mathtt{U}\,\psi)$, *and* $\mathtt{A}(\phi\,\mathtt{U}\,\psi)$ *are formulas if ϕ and ψ are.*

CTL semantics is given with respect to Kripke structures. In the following inductive definition of the semantics of CTL, $\mathcal{K}, q \models \psi$ denotes that ψ holds in the state q of the Kripke structure $\mathcal{K}$.

Definition 2.5 (CTL Semantics) *Given a Kripke structure $\mathcal{K} = \langle S, R\rangle$, the semantics of CTL formulas are inductively defined as follows*

- $\mathcal{K}, q_0 \models P$ *iff* $q_0 \in P$,
- $\mathcal{K}, q_0 \models \neg\psi$ *iff* $\mathcal{K}, q_0 \not\models \psi$,
- $\mathcal{K}, q_0 \models \mathtt{E}(\phi\,\mathtt{U}\,\psi)$ *iff there exists a path* $q_0q_1\cdots$ *and* $i \geq 0$ *such that* $\mathcal{K}, q_i \models \psi$ *and, for all* $0 \leq j < i$, $\mathcal{K}, q_j \models \phi$,
- $\mathcal{K}, q_0 \models \mathtt{A}(\phi\,\mathtt{U}\,\psi)$ *iff for all paths* $q_0q_1\cdots$ *there exists* $i \geq 0$ *such that* $\mathcal{K}, q_i \models \psi$ *and, for all* $0 \leq j < i$, $\mathcal{K}, q_j \models \phi$.

We will use three abbreviations

$$\begin{aligned} \mathtt{AF}\,\psi &\equiv \mathtt{A}(S\,\mathtt{U}\,\psi), &(2.3)\\ \mathtt{EF}\,\psi &\equiv \mathtt{E}(S\,\mathtt{U}\,\psi), &(2.4)\\ \mathtt{AG}\,\psi &\equiv \neg\mathtt{EF}\neg\psi. &(2.5)\end{aligned}$$

F stands for “future” or “eventually”. Since S is the complete set of states in the Kripke structure, the CTL formula S holds in any state. Thus, $\mathtt{AF}\,\psi$ means that for all execution paths, a state where ψ holds will eventually be reached. Similarly, $\mathtt{EF}\,\psi$ means that there exists an execution path reaching a state where ψ holds. G stands for “globally” or “always”. The formula $\mathtt{AG}\,\psi$ holds, if every state on any execution path satisfies ψ.

Example 2.3 Let $\mathcal{K}$ denote the Kripke structure shown in Figure 2.1. Since four of the transitions form a cycle $CABDC$, we have that from any state visited on an execution path produced from C, C can be reached. Thus, $\mathcal{K}, C \models \mathtt{AGEF}\,\{C\}$. ♢

We will often consider CTL formulas on execution trees produced from a set of states. To simplify the presentation, we therefore introduce the short notation

$$\mathcal{K}, Q \models \psi \equiv \forall q \in Q \,.\, \mathcal{K}, q \models \psi. \tag{2.6}$$

2.2 Binary Decision Diagrams

A reduced ordered Binary Decision Diagram (BDD) is rooted Directed Acyclic Graph (DAG) representing a Boolean function on a set of linearly ordered Boolean variables. It has one or two terminal nodes labeled 1 or 0, and a set of variable nodes. Each variable node is associated with a Boolean variable and has two outgoing edges *low* and *high*. Given an assignment of the variables, the value of the Boolean function is determined by a path starting at the root node and recursively following the high edge, if the associated variable is true, and the low edge, if the associated variable is false. The function value is $true$, if the label of the reached terminal node is 1; otherwise it is $false$. The graph is ordered such that all paths in the graph respect the ordering of the variables.

Example 2.4 A BDD representing the function $f(x_1, x_2) = x_1 \vee \neg x_1 \wedge \neg x_2$ is shown in Figure 2.3. ♢

A BDD is reduced such that no two distinct nodes u and v have the same variable name and low and high successors (Figure 2.4a), and no variable node u has identical low and high successors (Figure 2.4b). Due to these reductions, the number of nodes in a BDD of

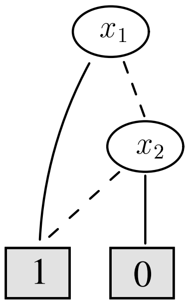


Figure 2.3: A BDD representing the function $f(x_1, x_2) = x_1 \vee \neg x_1 \wedge \neg x_2$. High and low edges are drawn with solid and dashed lines, respectively.

a regularly structured function is often much smaller than the number of truth assignments of the function. In particular, it can be shown that the size of a BDD representing any *symmetric function* only grows polynomially with the number of variables of the function [26].

Definition 2.6 *A Boolean function $f \in \mathbb{B}_n$ is called symmetric if each permutation ρ of the variables does not change the function value, i.e.,*

$$f(x_1, \cdots, x_n) = f(x_{\rho(1)}, \cdots, x_{\rho(n)}).$$

Another advantage is that the reductions make BDDs canonical. Large space savings can be obtained by representing a collection of BDDs in a single multirooted graph where the subgraphs of the BDDs are shared. Due to the canonicity, two BDDs are identical if and only if they have the same root. Consequently, when using this representation, equivalence check between two BDDs can be performed in constant time. In addition, BDDs are easy to manipulate. Any Boolean operation $f \star g$ on two BDDs f and g can be carried out in $O(|f||g|)$. The size of a BDD can depend critically on the variable ordering. To find an

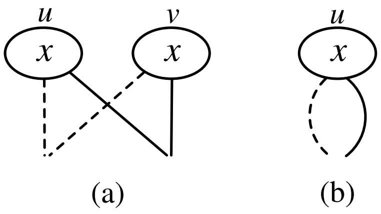


Figure 2.4: Reductions of BDDs. (a) nodes associated to the same variable with equal low and high successors will be converted to a single node. (b) nodes causing redundant tests on a variable are eliminated.

optimal ordering is a co-NP-complete problem in itself [26], but as illustrated in Example 2.5, a good heuristic for choosing an ordering is to locate variables close to each other if knowledge about their truth assignment removes a lot of uncertainty about the truth value of the Boolean function. [40].

Example 2.5 For the function $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$ there is an exponential difference between the size of a BDD with the variable ordering $x_1, y_1, \cdots, x_n, y_n$ and a BDD with the variable ordering $x_1, \cdots, x_n, y_1, \cdots, y_n$. For the latter ordering, the lack of information about the assignment of the y variables in the top section of the BDD, where the x variables are tested, causes an exponential growth of the graph [119]. The problem is illustrated for $n = 3$ in Figure 2.5. $\Diamond$

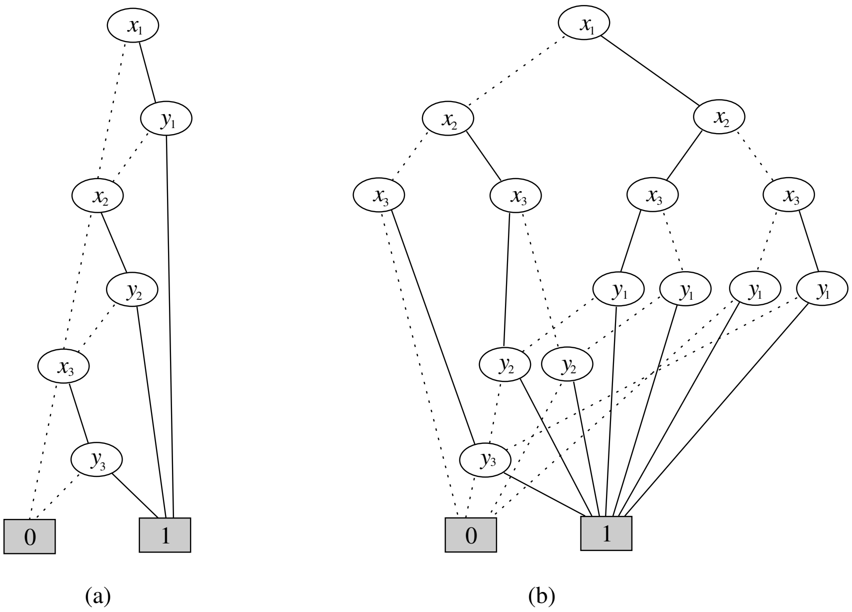


Figure 2.5: Two BDDs representing the function $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$. The BDD in (a) only grows linearly with the number of variables in the expression, while the BDD in (b) grows exponentially.

For a comprehensive introduction to BDDs and *branching programs* in general, we refer the reader to Bryant's original paper [26] and the books [121, 168].

BDD Packages

A BDD package is a collection of efficient data structures and algorithms for representing and computing basic operations on BDDs. Modern BDD packages (e.g.,[158, 112]) typically share the following common implementation features based on [25, 146]: 1) a single shared BDD with several roots representing a set of BDDs, 2) a set of dynamic programming algorithms for carrying out operations on the BDDs that due to a large number of distinct subproblems use a cache instead of a memoization table, and 3) data structures that facilitate dynamic variable reordering and garbage collection of unreferenced BDD nodes that is invoked when the percentage of unreferenced BDD nodes exceeds a preset threshold. The three major parameters are: the initial number of nodes allocated to the shared BDD, the cache size, and the type of dynamic variable reordering, if any.

2.3 Symbolic Model Checking

Model checking (e.g., [40]) is a subfield of computer science that applies efficient search procedures to determine if a finite transition system fulfills its specification. In other words, the transition system is *checked* to see whether it is a *model* of its specification. Given a Kripke structure $\mathcal{K}$ representing the system, a specification is a CTL formula that must hold in the initial state s_0 of the system. For our purpose, it is sufficient only to consider invariant specifications $\mathcal{K}, s_0 \models \texttt{AG}\, I$ where all states reachable from s_0 must be within the set I. An invariant specification is verified by performing a *reachability analysis* to find the set of states R reachable from s_0. It is then checked whether R is a subset of I. Obviously, an exhaustive explicit exploration faces the state space explosion problem. In 1987 McMillan suggested to use BDDs to search the state space implicitly to address this problem. He coined the technique *symbolic model checking* [119].

The basic idea in symbolic model checking is to use a BDD to represent the *characteristic function* of a set of states and the transition relation. Given a set A, its characteristic function $A(x) \equiv x \in A$ is a Boolean function identifying all elements of A. This is an *implicit* representation of A since the size of the BDD representing the characteristic function of A does not necessarily grow linearly with the cardinality of A. Due to the isomorphism between set algebra and Boolean algebra union, intersection, and complement of sets correspond to disjunction, conjunction and negation of their characteristic functions. In the sequel, we will not distinguish between set operations and their corresponding Boolean operations. Given a Kripke structure $\mathcal{K} = \langle S, R \rangle$, we can use a vector of Boolean *state variables* $\vec{v} = (v_1, \cdots, v_{\lceil \log(|S|) \rceil})$ to represent a state. Any subset of states Q can be represented by a Boolean function $Q(\vec{v})$ that can be encoded as a BDD. Similarly, a Boolean function $R(\vec{v}, \vec{v}\,')$, where unprimed and primed variables denote current and next states, can be used to represent the characteristic function of the transition relation.

Example 2.6 The Kripke structure in Figure 2.1 can be represented with two state variables $\vec{v} = (v_1, v_2)$ such that $A = (\mathit{false}, \mathit{false})$, $B = (\mathit{true}, \mathit{false})$, $C = (\mathit{false}, \mathit{true})$, and $D = (\mathit{true}, \mathit{true})$. The characteristic function of the transition relation is then

$$\begin{array}{rclcll} R(\vec{v}, \vec{v}\,') & = & \neg v_1 \wedge \neg v_2 & \wedge & v_1' \wedge \neg v_2' & \vee \\ & & v_1 \wedge \neg v_2 & \wedge & v_1' \wedge v_2' & \vee \\ & & \neg v_1 \wedge v_2 & \wedge & \neg v_1' \wedge \neg v_2' & \vee \\ & & \neg v_1 \wedge v_2 & \wedge & v_1' \wedge v_2' & \vee \\ & & v_1 \wedge v_2 & \wedge & \neg v_1' \wedge v_2'. & \end{array}$$

$\Diamond$

The crucial idea in symbolic model checking is to compute previous and next states via BDD operations. The next states of a set of states C, can be found by computing the *image* of C

$$\textsc{Img}(C) \;\equiv\; \Big(\exists \vec{v}\,.\; C(\vec{v}) \wedge R(\vec{v}, \vec{v}\,')\Big)[\vec{v}\,'/\vec{v}]. \tag{2.7}$$

Existential quantification is used to abstract the source state. The input to $\textsc{Img}(C)$ is the characteristic function of C in current state variables $C(\vec{v})$. The output is the characteristic function in current state variables of the states that can be reached by a single transition from C. Notice that such states may lie within C. The output is given in current state variables in order to use it as input for subsequent image computations. The reachable states from C can be computed by composing images from C until a fixed point is found. All Boolean functions in the image computation are represented by BDDs, and all Boolean operations are carried out directly on these BDDs. In the sequel, we will not distinguish between Boolean operations and their corresponding BDD operations.

Example 2.7 For the state C in the Kripke structure shown in Figure 2.1, we have $C(\vec{v}) = \neg v_1 \wedge v_2$. The image of C is given by

$$\begin{array}{rcl} \textsc{Img}(C) & = & \Big(\exists \vec{v}\,.\; (\neg v_1 \wedge v_2) \wedge R(\vec{v}, \vec{v}\,')\Big)[\vec{v}\,'/\vec{v}] \\ & = & \Big(\exists \vec{v}\,.\; \neg v_1 \wedge v_2 \wedge \neg v_1' \wedge \neg v_2' \;\vee\; \neg v_1 \wedge v_2 \wedge v_1' \wedge v_2'\Big)[\vec{v}\,'/\vec{v}] \\ & = & \Big(\neg v_1' \wedge \neg v_2' \;\vee\; v_1' \wedge v_2'\Big)[\vec{v}\,'/\vec{v}] \\ & = & \neg v_1 \wedge \neg v_2 \;\vee\; v_1 \wedge v_2. \end{array}$$

Thus, as expected, we get $\textsc{Img}(C) = \{A, D\}$. $\Diamond$

Previous states of a set of states C can be found by a similar computation called the *preimage* of C

$$\textsc{PreImg}(C) \;\equiv\; \exists \vec{v}\,'\,.\; R(\vec{v}, \vec{v}\,') \wedge C(\vec{v})[\vec{v}/\vec{v}\,']. \tag{2.8}$$

Again, the input is the characteristic function of C in current state variables. The output is the characteristic function (in current state variables) of the states from which a state in C can be reached by a single transition.

2.3.1 Partitioning

A common problem when computing the image and preimage is that the intermediate BDDs tend to be large compared to the BDD representing the result. Another problem is that the transition relation may grow very large if represented by a single BDD (a *monolithic* transition relation). In symbolic model checking, one of the most successful approaches to solve this problem is *transition relation partitioning* [28]. The technique relies on the observation that a system often can be characterized as either *asynchronous* with interleaved activity or *synchronous* with simultaneous activity. Consider the system model shown in Figure 2.6. For each computation, the state variables $v_1, \cdots, v_n$ are updated. Assume that

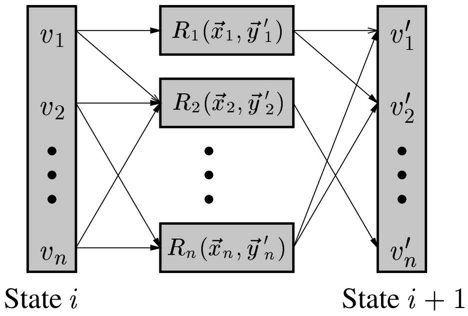


Figure 2.6: System model.

activity i computes the next value of the state variables in $\vec{y}_i$ given the current value of the state variables in $\vec{x}_i$ and is characterized by the transition relation $R_i(\vec{x}_i, \vec{y}'_i)$. If the system is asynchronous, only a single subsystem is active in a computation step and only the next state variables of this subsystem change value. Otherwise, if the system is synchronous, each subsystem is active in a computation step and calculates a new value of its associated state variables. In this case, we must assume that the sets of variables changed by the subsystems form a partitioning of the state variables. Let Z denote the state variables in vector $\vec{z}$. In the asynchronous case, the total transition relation is

$$R(\vec{v}, \vec{v}\,') = \bigvee_{i=1}^{n} \Big(R_i(\vec{x}_i, \vec{y}'_i) \wedge \bigwedge_{v \notin Y_i} (v' \Leftrightarrow v) \Big) \tag{2.9}$$

while in the synchronous case, it is

$$R(\vec{v}, \vec{v}\,') = \bigwedge_{i=1}^{n} R_i(\vec{x}_i, \vec{y}'_i). \tag{2.10}$$

Thus, the transition relation can either be represented as a *disjunctive partitioning* or a *conjunctive partitioning* of subrelations. The main point about partitioning is that the complete transition relation never needs to be computed since both the image and preimage computations can be carried out directly on the subrelations. In the asynchronous case, we get

$$\textsc{Img}(C) \;=\; \bigvee_{i=1}^{n} \Big(\exists \vec{y}_i \,.\, C(\vec{v}) \wedge R_i(\vec{x}_i, \vec{y}\,'_i)\Big)[\vec{y}\,'_i/\vec{y}_i]. \tag{2.11}$$

We exploit that all variables except the ones modified by the active sybsystem are unchanged. Thus, no quantification over these variables is necessary. This often has a substantial positive effect on the complexity of the computation. The reason is that the complexity of quantification on BDDs may be exponential in the number of quantified variables.[3] In practice, it is often an advantage to merge some of the subrelations [143] and combine quantification and disjunction to a single BDD operation (e.g.,[112]). A similar approach can be used to simplify the preimage computation

$$\textsc{PreImg}(C) \;=\; \bigvee_{i=1}^{n} \exists \vec{y}\,'_i \,.\, R_i(\vec{x}_i, \vec{y}\,'_i) \wedge C(\vec{v})[\vec{y}_i/\vec{y}\,'_i]. \tag{2.12}$$

The conjunctive case is more complicated due to the fact that existential quantification does not distribute over conjunction. However, the subrelations can be moved out of scope of existential quantification if they do not depend on any of the variables being quantified. This technique is often referred to as *early quantification*. For the image computation, we get

$$\textsc{Img}(C) = \Big(\exists \vec{z}_n \,.\, (\cdots(\exists \vec{z}_1 \,.\, C(\vec{v}) \wedge R_1(\vec{x}_1, \vec{y}\,'_1))\cdots) \wedge R_n(\vec{x}_n, \vec{y}\,'_n)\Big)[\vec{v}'/\vec{v}] \tag{2.13}$$

where $Z_j \cap \bigcup_{i=j+1}^{n} X_i = \emptyset$ for $1 \leq j < n$ and $\bigcup_{i=1}^{n} Z_i = \{v_1, \cdots, v_n\}$.

Similarly, for the preimage computation, we have

$$\textsc{PreImg}(C) = \exists \vec{z}\,'_n \,.\, R_n(\vec{x}_n, \vec{y}\,'_n) \wedge (\cdots(\exists \vec{z}\,'_1 \,.\, R_1(\vec{x}_1, \vec{y}\,'_1) \wedge C(\vec{v})[\vec{v}/\vec{v}\,'])\cdots) \tag{2.14}$$

where $Z'_j \cap \bigcup_{i=j+1}^{n} Y'_i = \emptyset$ for $1 \leq j < n$ and $\bigcup_{i=1}^{n} Z'_i = \{v'_1, \cdots, v'_n\}$.

A large number of heuristics have been developed for choosing and arranging partitions in the conjunctive case (e.g., [143, 120]). The main idea is to avoid a blow up of the intermediate BDDs of the image and preimage computation by reducing the life span of variables. Assume that a variable is introduced in the computation by partition i and that the variable is removed again by the existential quantification associated with partition j. The life span of the variable is then $j - i$. Another approach to reduce the complexity of the

[3] From an artificial intelligence point of view, this simplified image computation is an efficient solution to the frame problem of asynchronous systems.

image and preimage computation is to compress the transition relation using an approach called *iterative squaring* [27]. The idea is to incrementally compute the closure of the transition relation. This computation, however, is often very complex. It is normally only an advantage if the domain has very high sequential depth (e.g., due to counters [61]).

2.3.2 Frontier Set Simplification

Partitioning is often combined with *frontier set simplification* [41]. The purpose of frontier set simplification is to reduce the size of the BDDs representing the frontier of bacward or forward search based on the image or preimage computation. Consider computing the set of states reachable from C using the image computation. The set of states R_1 that can be reached in one step or less is

$$R_1 = \textsc{Img}(C) \cup C.$$

Similarly, the set of states R_2 that can be reached in two steps or less is

$$R_2 = \textsc{Img}(R_1) \cup R_1.$$

This computation can be simplified by only computing next states from the frontier of the search

$$R_2 = \textsc{Img}(R_1 \setminus C) \cup R_1.$$

The set $F = R_1 \setminus C$ may have a large BDD representation. However, we can choose it anywhere in the range $R_1 \setminus C \subseteq F \subseteq R_1$ to obtain a small BDD. The research on frontier set simplification has developed several heuristic BDD operations for finding a good candidate for F.

2.3.3 Splitting

A more direct approach for computing the image and preimage of a set of states is to use a transition function. Consider for example a transition system with n state variables $v_1, \cdots, v_n$ and the transition function $f : \mathbb{B}^n \rightarrow \mathbb{B}^n$ given by

$$\begin{aligned} v'_1 &= f_1(v_1, \cdots, v_n) \\ &\cdots \\ v'_n &= f_n(v_1, \cdots, v_n). \end{aligned}$$

The image of a state $s \in \mathbb{B}^n$ is the mathematical image $f(s)$ of s. To find the image $f(S)$ of all states S in the domain (the unrestricted image) a technique called *input splitting* can be employed (e.g., [121]). It is based on the observation that the unrestricted image computation can be decomposed with respect to the input variables

$$f(S) = f|_{v_i \leftarrow true} \vee f|_{v_i \leftarrow false}.$$

The decomposition is carried out recursively until each restricted f function has constant elements. The characteristic function of the image (in next state variables) is given by the resulting expression. The approach can be extended to an arbitrary set of states and is particularly efficient for problems where it is impossible to arrange a conjunctive partitioning to allow an efficient early quantification [123].

2.3.4 BDD Package Adjustment

An invariant specification is checked by performing a sequence of image computations until the set of covered states reaches fixed point. Experimental studies indicate that the BDD package parameters should be adjusted differently for BDD-based model checking compared to circuit verification where a BDD representing a digital circuit is compared to a BDD representing its specification [177]. The experiments show that model checking computations have a large number of repeated subproblems across the top level operations. Thus, a large cache size is more important for model checking than for circuit verification. Furthermore, model checking computations can have a very high death and rebirth rate (unreferenced nodes being referenced again) compared to circuit computations. Thus, garbage collection should occur less frequently, which for example can be accomplished by initially allocating a large number of nodes for the shared BDD. Finally, dynamic reordering of variables is efficient given an initial bad variable ordering, but given a good initial variable ordering the time spend on reordering does not pay off.

2.4 Heuristic Search

A classical search problem is similar to a deterministic planning problem defined in Definition 3.2. The only difference is that actions are associated with a positive cost.[4] Let the function $c : Act \to \mathbb{R}^+$ define action costs. A solution to a classical search problem is a deterministic plan $\pi = a_1 \cdots a_n$. The *length* of π is n. The *cost* of π is

$$\textsc{Cost}(\pi) \equiv \sum_{i=1}^{n} c(a_i). \tag{2.15}$$

[4]The cost of an action must be positive since we require that infinite paths have unbounded total cost.

Definition 2.7 (Optimal Search Problem Solution) *A search problem solution π to search problem is optimal if it has minimum cost.*

Example 2.8 The deterministic planning problem shown in Figure 3.1 can be extended to a search problem by adding action costs as shown in Figure 2.7. We have

$$\begin{aligned}
S &= \{A, B, C, D\}, \\
Act &= \{\alpha, \beta, \gamma\}, \\
\rightarrow &= \{\langle A, \beta, B\rangle, \langle B, \gamma, D\rangle, \langle C, \alpha, A\rangle, (C, \beta, D\rangle, \langle D, \alpha, C\rangle\}, \\
s_0 &= C, \\
G &= \{B\}, \\
c &= \{\alpha \mapsto 1.0, \beta \mapsto 1.0, \gamma \mapsto 2.0\}.
\end{aligned}$$

An optimal solution is $\alpha\beta$ with a length and cost of 2. $\Diamond$

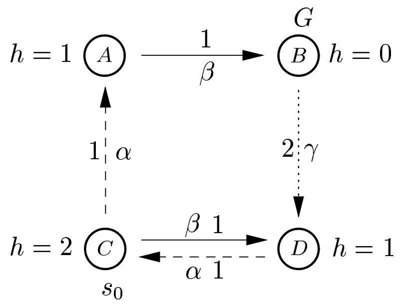


Figure 2.7: A search problem derived from the deterministic planning problem discussed in Example 3.1. The h-values associated with each state defines a heuristic function used in Example 2.9.

Classical search algorithms like A* and pure heuristic search are characterized by building a search tree superimposed over the state space during the search process. Each *search node* in the tree is a pair $\langle s, I\rangle$ where s is a single state and $I \in \mathbb{R}^d$ is a d-dimensional vector of real numbers representing information associated with the node to guide the search. The root of the search tree contains the initial state. We will assume that the initial state belongs to a search node with node information I_0. The leaf nodes of the tree correspond to states that do not have successors in the tree, either because they have not been expanded yet, or because they were expanded, but had no children. In each step, the search algorithm chooses one leaf node to expand. The collection of unexpanded nodes is called the *fringe* or *frontier* of the search tree. It is important to distinguish between the search domain and the search tree. For finite but cyclic search domains, the search tree may be infinite. Best-First Search (BFS) describes a collection of search algorithms, each of which has a cost associated with each node and at each node expansion cycle chooses a node of lowest cost to expand next. Depending on the particular cost function chosen, we get different BFS algorithms. To lower the complexity of the node selection, the frontier is often implemented as a priority queue on the node costs. Figure 2.8 shows the BFS algorithm. The solution extraction function in line 5 simply obtains a solution by tracing back the actions from the goal node to the root node. The EXPAND function in line 6 finds the set of child nodes of a single node, and ENQUEUEALL inserts each child in the frontier queue.

function BFS(s_0, I_0, G)
1 $\mathit{frontier} \leftarrow$ MAKEQUEUE$(\langle s_0, I_0 \rangle)$
2 **loop**
3   **if** $|\mathit{frontier}| = 0$ **then return** "no solution exists"
4   $\langle s, I \rangle \leftarrow$ REMOVETOP$(\mathit{frontier})$
5   **if** $s \in G$ **then return** ExtractSolution$(\mathit{frontier}, \langle s, I \rangle)$
6   $\mathit{frontier} \leftarrow$ ENQUEUEALL$(\mathit{frontier}$, EXPAND$(\langle s, I \rangle))$

Figure 2.8: The Best-First Search (BFS) algorithm.

The A* algorithm is probably the best known and most well studied of the BFS algorithms. A* sorts the unexpanded nodes in the priority queue in ascending order given by a *heuristic evaluation function* f. The evaluation function is defined by

$$f(n) = g(n) + h(n) \tag{2.16}$$

where $g(n)$ is the cost of the path in the search tree leading from the root node to n, and $h(n)$ is a *heuristic function* estimating the cost of a minimum cost path leading from the state in n to some goal state. Thus $f(n)$ measures the minimum cost over all solution paths constrained to go through the state in n.

Example 2.9 The search tree built by A* for the problem introduced in Example 2.8 and the heuristic function defined in Figure 2.7 is shown in Figure 2.9. $\Diamond$

The properties of A* have been surveyed by Pearl [130]. A* is *sound* and *complete*, since the node expansion operation is assumed to be correct, and infinite cyclic paths have unbounded cost. A* further finds optimal solutions if the heuristic function $h(n)$ is *admissible*, that is, $h(n)$ is a lower bound estimate such that $h(n) \leq h^*(n)$ for all n, where $h^*(n)$ is the minimum cost of a path going from the state in n to a goal state. A* is *optimally efficient* for any admissible heuristic function. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A* [44]. It can be shown that every node on the

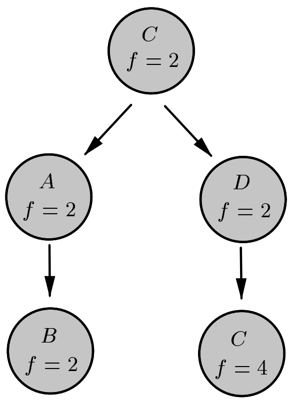


Figure 2.9: Search tree example.

frontier with $f(n) < C^*$, where C^* is the optimal cost, eventually will be expanded by A*. Thus, the complexity of A* is directly tied to the accuracy of the estimates provided by $h(n)$. When A* employs a perfectly informed heuristic ($h(n) = h^*(n)$), it is guided directly toward the closest goal. At the other extreme, when no heuristic at all is available ($h(n) = 0$), the search becomes exhaustive, normally yielding exponential complexity. In general, A* has linear complexity if the absolute error of the heuristic function is constant, but it may have exponential complexity if the relative error is constant. Subexponential complexity requires that the growth rate of the error is logarithmically bounded [147]

$$|h(n) - h^*(n)| \in O(\log h^*(n)).$$

The complexity results are disappointing due to the fact that practical heuristic functions often are based on a relaxation of the search problem that causes $h(n)$ to have constant or near constant relative error. The results show that practical application of A* still may be very search intensive. Often better performance of A* can be obtained by weighting the g and h-component of the evaluation function [140]

$$f(n) = (1 - w)g(n) + wh(n), \text{ where } w \in [0, 1]. \tag{2.17}$$

Weighted A* can be used to implement a wide range of BFS algorithms. Weights $w = 0.0, 0.5$, and 1.0 correspond to uniform cost search (Dijkstra's algorithm), A*, and pure heuristic search, respectively. Weighted A* is optimal in the range $[0.0, 0.5]$ but often finds solutions faster in the range $(0.5, 1]$.

Another drawback of A* is that its space complexity is very high due to the explicit representation of the search tree. For that reason an iterative deepening version of A* called IDA* has been developed [107]. This algorithm has space complexity linear with the search depth. However, unless the search domain is a tree, it may perform a highly redundant search.

2.5 Summary

This chapter has described Quantified Boolean Formulas (QBF) as a concise logic for representing the complex Boolean operations involved in BDD-based planning. To specify non-deterministic plans, we have presented Kripke structures and the Computation Tree Logic (CTL). We then spend two sections describing the key features of the Binary Decision Diagram (BDD) and the techniques developed in model checking to represent and search a state space efficiently with BDDs. Finally, we have described classical heuristic search algorithms and heuristic search techniques.

Chapter 3

BDD-Based Planning

In this chapter, we describe several basic encoding and search techniques for deterministic and non-deterministic BDD-based planning. Section 3.1 defines deterministic planning and presents three principles for encoding STRIPS planning problems with BDDs. Moreover, it describes a general BDD-based bidirectional breadth-first search algorithm for generating deterministic plans. Section 3.2 introduces the definition of non-deterministic planning used in the thesis and shows how to encode NADL and NADL$^+$ planning problems with BDDs. Finally, it introduces a general BDD-based backward breadth-first search algorithm for generating weak, strong cyclic, and strong non-deterministic plans.

3.1 Deterministic Planning

Classical AI-planning considers domains with a finite set of states and a finite set of deterministic actions.

Definition 3.1 (Deterministic Planning Domain) *A deterministic planning domain is a tuple $\langle S, Act, \rightarrow \rangle$ where S is a finite set of states, Act is a finite set of actions, and $\rightarrow \subseteq S \times Act \times S$ is a deterministic transition relation of action effects. Instead of $(s, a, s') \in \rightarrow$, we write $s \xrightarrow{a} s'$.*

The transition relation $\rightarrow$ is deterministic if actions can lead to at most one possible next state. That is

$$s \xrightarrow{a} p' \wedge s \xrightarrow{a} q' \;\Rightarrow\; p' = q'.$$

An action a is *applicable* in a state s iff $s \xrightarrow{a} s'$ for some state s'. A deterministic planning problem is given by a single initial state and a set of goal states.

Definition 3.2 (Deterministic Planning Problem) *A deterministic planning problem is a tuple $\langle \mathcal{D}, s_0, G \rangle$ where $\mathcal{D}$ is a deterministic planning domain, $s_0 \in S$ is an initial state, and $G \subseteq S$ is a set of goal states.*

A solution to (or *plan* for) a deterministic planning problem is a sequence of actions forming a path from the initial state to a goal state.

Definition 3.3 (Deterministic Plan) *Let $\mathcal{P}$ be a deterministic planning problem. A solution or plan for $\mathcal{P}$ is a sequence of actions $\pi = a_1 \cdots a_n$ such that there exists a path $q_0 \cdots q_n$ where $q_0 = s_0$, $q_n \in G$, and for $0 < i \leq n$, we have $q_{i-1} \stackrel{a_i}{\to} q_i$.*

The *length* of a plan $a_1 \cdots a_n$ is n. A plan is *optimal* if it has minimum length.

Example 3.1 The deterministic planning problem shown in Figure 3.1 has

$$\begin{aligned} S &= \{A, B, C, D\}, \\ Act &= \{\alpha, \beta, \gamma\}, \\ \to &= \{\langle A, \beta, B\rangle, \langle B, \gamma, D\rangle, \langle C, \alpha, A\rangle, (C, \beta, D\rangle, \langle D, \alpha, C\rangle\}, \\ s_0 &= C, \\ G &= \{B\}. \end{aligned}$$

An optimal plan solving the problem is $\alpha\beta$ and has length 2. $\Diamond$

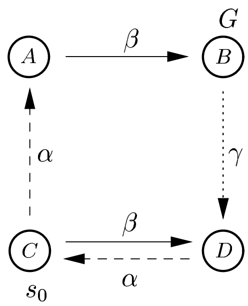


Figure 3.1: A deterministic planning problem with four states A, B, C, and D and three actions α (dashed), β (solid), and γ (dotted). The initial state is C and the set of goal states is a singleton set $\{B\}$.

3.1.1 Encoding STRIPS Domains

Classical deterministic planning problems are often written in planning languages such as STRIPS [58], ADL [132], and PDDL [118]. In these languages, states are represented by conjunctions of function-free ground predicates.[1] A STRIPS planning domain is a pair $\mathcal{D} = \langle P, A \rangle$ where P is a set of predicates and A is a set of action schemas. Each action schema is a tuple $\langle par, pre, add, del \rangle$ where par is a set of parameter variables, and pre, add, and del are sets of predicates from P where the only free variables are parameter variables in par.

Example 3.2 The Gripper domain used in the AIPS-98 planning competition [113] is shown in Figure 3.2. In this domain there are three actions schemas and 7 predicates of which three are used to indicate the type of objects. $\diamondsuit$

Predicates

room(R)
ball(B)
gripper(G)
at−robby(R)
at(B,R)
free(G)
carry(O,G)

Actions

Move
par: *FROM, TO*
pre: *room(FROM), room(TO), atRobby(FROM)*
add: *atRobby(TO)*
del: *atRobby(FROM)*

Pick
par: *OBJ, ROOM, GRIPPER*
pre: *ball(OBJ), room(ROOM), gripper(GRIPPER), at(OBJ, ROOM), atRobby(ROOM), free(GRIPPER)*
add: *carry(OBJ, GRIPPER)*
del: *at(OBJ, ROOM), free(GRIPPER)*

Drop
par: *OBJ, ROOM, GRIPPER*
pre: *ball(OBJ), room(ROOM), gripper(GRIPPER), carry(OBJ, GRIPPER), atRobby(ROOM)*
add: *at(OBJ, ROOM), free(GRIPPER)*
del: *carry(OBJ, GRIPPER)*

Figure 3.2: The Gripper domain of the AIPS-98 planning competition. A robot called Robby has grippers to move objects between rooms. The Move action moves Robby between rooms, while the Pick and Drop actions load and unload objects into a gripper.

A STRIPS planning problem is a tuple $\langle \mathcal{D}, O, I, G \rangle$ where $\mathcal{D}$ is a STRIPS planning domain, O is a set of constant terms forming the objects of the problem, I is a set of ground

[1] More elaborate state representations exist, but representing states as sets of ground predicates is the main idea.

predicates that are true in the initial state (all other ground predicates are assumed to be false initially), and G is a set of ground predicates that must be true in a goal state (all other ground predicates can have arbitrary truth values). The actions of the problem are generated from the action schemas by instantiating the parameters with objects of the problem. In a given state S, an action $\langle pre, add, del\rangle$ is applicable if $pre \subseteq S$, and the resulting state is $S' = (S \cup add) \setminus del$.

Example 3.3 The objects and initial state of the Gripper planning problem shown in Figure 3.3 specifies that there are two rooms (rooma,roomb), two grippers (left,right), and that the movable object is a ball (ball1). Initially, both Robby and the ball is in room A and the goal is to move the ball to room B. ◊

Objects

rooma, roomb, ball1, left, right

Initial

room(rooma), room(roomb), ball(ball1), atRobby(rooma), free(left), free(right) at(ball1, rooma) gripper(left), gripper(right)

Goal

at(ball1, roomb)

Figure 3.3: A Gripper planning problem, with two rooms, one ball object to move, and two grippers on Robby. Initially, both Robby and the ball are in room A, and the goal is to move the ball to room B.

For classical deterministic planning problems described in a STRIPS like language, BDD-based deterministic planning involves two orthogonal problems. The first is to represent the planning domain compactly. The second is to compute a BDD representation of the transition relation and perform a BDD-based exploration of the state space.

Representing STRIPS Domains Compactly

Consider the STRIPS description of the Gripper problem in Example 3.3. A simple way to represent the transition relation of this problem with a BDD is to ground all predicates and action schemas and use a Boolean state variable to represent each ground predicate. However, this often results in a very redundant encoding that is impossible to handled efficiently with BDDs. In order to encode STRIPS domains efficiently, we follow mainly [50] and use three principles to compress the domain description.

Principle 1. The first principle is to remove predicates from the domain description that do not change their truth-value. These predicates are called *static predicates*. They are typically used to represent typing information like the static predicates $room(R)$, $ball(B)$, and $gripper(G)$ in the Gripper problem. It is simple to identify static predicates by checking that they are not included in a delete or add set of any action. Moreover, since we already now their truth value by inspecting the initial state, they can be abstracted away in a compressed encoding of the transition relation.

Principle 2. The second principle is to find the domain of predicate arguments and action schema parameters which is often a small subset of the total set of objects O. The static predicates may provide information to restrict these domains, but there can be constraints on the domains that only can be found by finding the set of reachable states from the initial state. Computing these states is as hard as solving the planning problem itself. Instead, the set of reachable states can be approximated by a relaxed reachability analysis where the delete set of actions is ignored. This estimate will always include the reachable states. If implemented carefully, the analysis can be carried out in a small fraction of the time needed to solve the complete planning problem [50].

Principle 3. The third principle is to use numerical state variables instead of predicates to represent locations of objects. Predicates often encode physical locations of objects. In the Gripper problem, the four predicates $at(ball1, rooma)$, $at(ball1, roomb)$, $carry(ball1, left)$, and $carry(ball1, right)$ encode the four possible locations of the ball. However, since the ball at most can be at one location at a time, we only need a single numerical state variable represented by $\log(4) = 2$ bits to represent the truth value of these predicates. Sets of predicates with this property are called *single-valued* [64] or *balanced* [50]. Balanced predicates can be found automatically by generating candidate sets of predicates and proving by induction that they are balanced. The base case of this proof is to show that they are balanced in the initial state. The inductive step is to show that each action preserves their balance [141].

When given a planning problem defined in the STRIPS part of the PDDL language, the BIFROST search engine described in Appendix A can perform these three analysis steps automatically. The complete analysis can normally be carried out in a small fraction of the total time needed to solve the planning problem.

Encoding STRIPS Domains with BDDs

In order to compute a BDD representation of the transition relation, we first observe that a deterministic planning domain is an asynchronous system in the sense that only a single action is active in each step. Thus, as described in Section 2.3.1, a disjunctive partitioning of the transition relation can be used to lower the complexity of the image and preimage computation. For each action i in the compressed encoding of the domain, we get a subrelation

$$R_i(\vec{x}_i, \vec{y}'_i) = \bigwedge_{x \in pre_i} x \;\wedge\; \bigwedge_{y \in add_i} y' \;\wedge\; \bigwedge_{y \in del_i} \neg y'. \tag{3.1}$$

To conform to the definition of the transition relation, the subrelation should contain information about which action the transitions are associated with. However, as described below, we can use the partitioning itself to hold this information when extracting the actions of a solution. Saving variables in the encoding of the transition relation is important for keeping the complexity of the image and preimage computations low.

Due to the large number of small BDDs representing the action relations, it is almost always an advantage to combine them into larger partitions. Care must be taken to merge the subrelations such that partitions that only modify a small subset of variables are produced. It is hard to produce optimal solutions to this problem. However, an approximation that works satisfactory in practice is to sort the subrelations according to which variables they modify and merge them from left to right according to a threshold on the size of the BDD representing the resulting partition. Typical "good" values of the threshold is in the range 5000 to 10000 BDD nodes. As described in Section 2.2, the size of a BDD is sensitive to the variable ordering. In general, related variables should be close to each other in the ordering. Since the current and next state variables of a state variable almost always are highly dependent, it is often beneficial to interleave them in the ordering. However, no previous work has addressed how the state variables of planning domains should be ordered. Heuristics like *fan in* and *weight* for constructing good variable orders of combinational circuits [121] still need to be developed for BDD-based planning. In practice, the natural ordering of state variables of a planning domain often turns out to be efficient, since it reflects the semantics of the variables. Otherwise, the dynamic re-ordering techniques of the BDD package can be used to find good orderings.

3.1.2 Planning Algorithms

Given a BDD representation of the transition relation, it is simple to use the image and preimage computation to implement optimal breadth-first forward, backward, and bidirectional search. The forward and backward search algorithms are special cases of the bidirectional search algorithm shown in Figure 3.4. In each iteration, the algorithm either computes the frontier states in forward or backward direction. The set $reached$ contains all explored states and is used to prune a new frontier from previously visited states. If the set of pruned frontier states is empty, the algorithm returns "no solution exists". If an overlap between the forward and backward search frontier is found, the algorithm extracts and returns a solution. Otherwise the search continues. A good heuristic for deciding in which

function BIDIRECTIONAL BREADTH-FIRST SEARCH(s_0, G)

$reached \leftarrow \emptyset$
$forwardFrontier_0 \leftarrow \{s_0\}$; $i \leftarrow 0$
$backwardFrontier_0 \leftarrow G$; $j \leftarrow 0$
while $forwardFrontier_i \cap backwardFrontier_j = \emptyset$
  if TIME$(forwardFrontier_i) <$ TIME$(backwardFrontier_j)$
    $i \leftarrow i + 1$
    $forwardFrontier_i \leftarrow$ IMG$(forwardFrontier_{i-1}) \setminus reached$
    $reached \leftarrow reached \cup forwardFrontier_i$
    if $forwardFrontier_i = \emptyset$ **return** "no solution exists"
  else
    $j \leftarrow j + 1$
    $backwardFrontier_j \leftarrow$ PREIMG$(backwardFrontier_{j-1}) \setminus reached$
    $reached \leftarrow reached \cup backwardFrontier_j$
    if $backwardFrontier_j = \emptyset$ **return** "no solution exists"
return EXTRACTSOLUTION$(forwardFrontier, backwardFrontier)$

Figure 3.4: BDD-based Bidirectional Breadth-First Search.

direction to expand the search is simply to choose the direction where the previous frontier took least time to compute [51]. When using this heuristic, bidirectional search has similar or better performance than both forward and backward search, since it will adapt to one of these algorithms if the frontiers always are faster to compute in a particular direction. The complexity of extracting a solution is normally much lower than the complexity of computing the sequence of expansions of the search frontier. To realize this, consider having found an overlap between the forward and backward search frontier. For each state s in the overlap, there exists an optimal solution passing through s. Consequently, we can pick a single state in the overlap and trace its associated optimal solution. To find the part of the solution from s to a goal state, images of s are intersected with the backward search frontiers. For each of these computations, the image only needs to be computed for a single state. This can be done very fast relative to the time needed for computing images during search since the BDD of a single state is small. When performing these image computations a version of the transition relation is employed where no subrelations of actions have been merged. Since each subrelation is associated with a particular action, this transition relation can be used to extract the actions of the solution path. To extract the part of the plan leading from the initial state to s a similar sequence of preimage computations is carried out.

The growth rate of the search frontier is usually depending highly on the search direction. For most problems, the backward growth rate is substantially larger than the forward. The reason for this is that the states reached from the initial state always are legal states of the system modeled by the domain, while the states reached from the goal states may be illegal states of the system. Thus, a regular structure of the modeled system may only be reflected in the forward search frontier. The difference in growth rate often disappears if the set of goal states is reduced to legal system states.

A major problem of BDD-based planning is a high growth rate of BDDs representing the search frontier [86]. Frontier set simplification can be used to address this problem (see Section 2.3.1). However, the technique does not seem work well on typical planning problems.

Since the reachability analysis that forms the core of symbolic model checking resembles the state space search performed by the bidirectional search algorithm shown in Figure 3.4, we would expect that the BDD package parameters should be adjusted similarly for symbolic model checking and planning. No systematic experiments have been carried out to confirm this, but our experiences with deterministic planning problems fit well with the hypothesis that: 1) a planning problem initiated with a good variable order seems always to perform better without dynamic variable reordering, 2) each garbage collection seems to impair performance by deleting nodes that later must be recomputed, and 3) a too little cache can cause a performance degradation of several factors (a cache size that works well in practice is about 10 percent of the total number of allocated nodes). However, there also seems to be significant differences between typical model checking problems and planning problems. The BDDs representing the search frontier of a typical planning problem often grow very fast compared to the BDDs representing the search frontier of a typical symbolic model checking problem [119]. The reason seems to be that there are several subtle differences between typical verification problems and planning problems. First of all, planning problems tend to be combinatorially hard compared to formal verification problems. Verification often considers digital circuits and software descriptions that are large compared to the logical problem they contain. Planning problems, on the other hand, are normally fairly dense representations of a combinatorial problem. Second, the graph diameter of planning domains is often larger than the graph diameter of the domains studied in formal verification. The reason is that planning problems normally involve sequencing a large number of actions, while symbolic model checking problems typically consider synchronous systems where a global state change can happen in each iteration.

3.2 Non-Deterministic Planning

A deterministic action can lead to at most one possible next state. A more general model is to assume that the outcome actions is uncertain such that actions may lead to one of several possible next states. For instance, when the robot in the Gripper domain picks a ball, it may be that it either succeeds and holds the ball in its gripper in the next state, or it fails and ends in a state where the ball still is on the floor and the gripper is empty. Markov decision processes (MDPs) model such non-determinism by defining the effect of actions as a probability distribution over the state space. We will consider a simpler model of non-determinism where the effect of an action is defined by a set of possible next states. Unless an alternative interpretation is clear from the context, the term *non-determinism* will be used to refer to this particular model.

Non-determinism can model a wide range of dynamic systems [3]. Common to all of them is that an active environment interact with the actions. The environment can for instance cause otherwise deterministic actions to fail, or it can control a subset of the actions. In the latter case these uncontrollable actions may either be interleaved or simultaneous with controllable actions. In both cases, it can be modeled by non-deterministic controllable actions.

A non-deterministic planning domain is similar to a deterministic planning domain except that the actions may be non-deterministic

Definition 3.4 (Non-Deterministic Planning Domain) *A non-deterministic planning domain is a tuple $\langle S, Act, \rightarrow \rangle$ where S is a finite set of states, Act is a finite set of actions, and $\rightarrow \subseteq S \times Act \times S$ is a non-deterministic transition relation of action effects. Instead of $(s, a, s') \in \rightarrow$, we write $s \xrightarrow{a} s'$.*

The set of next states of an action a applied in state s is given by

$$\textsc{Next}(s, a) \quad \equiv \quad \{s' \, : \, s \xrightarrow{a} s'\}. \tag{3.2}$$

An action a is called *applicable* in state s iff $\textsc{Next}(s, a) \neq \emptyset$. The set of applicable actions in a state s is given by

$$\textsc{App}(s) \quad \equiv \quad \{a \, : \, \textsc{Next}(s, a) \neq \emptyset\}. \tag{3.3}$$

A non-deterministic planning problem is similar to a deterministic planning problem.

Definition 3.5 (Non-Deterministic Planning Problem) *A non-deterministic planning problem is a tuple $\langle \mathcal{D}, s_0, G \rangle$[2] where $\mathcal{D}$ is a non-deterministic planning domain, $s_0 \in S$ is an*

[2]Several of the non-deterministic algorithms introduced in the thesis can handle uncertainty about the initial state represented by a set of initial states S_0. However, for the sake of simplicity of the presentation, we assume the initial state to be fully known.

initial state, and $G \subseteq S$ is a set of goal states.

A non-deterministic plan could be a sequence of actions that is guaranteed to reach a goal state regardless of the non-determinism of the domain. That is, for all uncertain action effects, the execution of the plan leads to a goal state. Such plans are called *conformant plans* [68, 35]. However, since conformant plans seldom exist, we define a non-deterministic plan to be a set of *state-action pairs* defined below.

Definition 3.6 (State-action pair (SA)) *Let $\mathcal{D}$ be a non-deterministic planning domain. A state-action pair $\langle s, a\rangle$ of $\mathcal{D}$ is a state $s \in S$ associated with an applicable action $a \in$ $\text{APP}(s)$.*

The set of SAs define a function from states to sets of actions relevant to apply in order to reach a goal state. This definition is identical to the state-action table definition used in [36, 37, 42, 34] and is similar to universal plans [154], policies in reinforcement learning (e.g.,[122]), and strategies in concurrent reachability games [43].

Definition 3.7 (Non-Deterministic Plan) *Let $\mathcal{D}$ be a non-deterministic planning domain. A non-deterministic plan for $\mathcal{D}$ is set of state-action pairs of $\mathcal{D}$.*

States are assumed to be fully observable. An execution of a non-deterministic plan is an alternation between observing the current state and choosing an action to apply from the set of actions associated with the state. Similar to policies in game theory, we call a non-deterministic plan *static* if there is at most a single action associated with each state. Otherwise, we call it *dynamic*, since an agent executing the plan may change its preference about which action to apply.

The set of states *covered* by a plan π is

$$\text{STATES}(\pi) \equiv \{s : \exists a \,.\, \langle s, a\rangle \in \pi\}. \tag{3.4}$$

The set of actions in a plan π associated with a state s is

$$\text{ACT}(\pi, s) \equiv \{a : \langle s, a\rangle \in \pi\}. \tag{3.5}$$

The *closure* of a plan π is the set of possible end states

$$\text{CLOSURE}(\pi) \equiv \{s' \notin \text{STATES}(\pi) : \exists \langle s, a\rangle \in \pi \,.\, s' \in \text{NEXT}(s, a)\}. \tag{3.6}$$

A plan π is said to be *total* iff $\text{CLOSURE}(\pi) \subseteq G$.

Example 3.4 A non-deterministic version of the deterministic planning problem described in Example 3.1 is shown in Figure 3.5. We have

$$\begin{aligned} S &= \{A, B, C, D\}, \\ Act &= \{\alpha, \beta, \gamma\}, \\ \rightarrow &= \{\langle A, \beta, B\rangle, \langle B, \gamma, D\rangle, \langle C, \alpha, A\rangle, (C, \alpha, D\rangle, \langle D, \beta, C\rangle\}, \\ s_0 &= C, \\ G &= \{B\}. \end{aligned}$$

Notice that the α action is non-deterministic since it can lead to two states from s_0. A plan for solving the problem could be $\pi = \{\langle C, \alpha\rangle, \langle D, \beta\rangle, \langle A, \beta\rangle\}$. We have $\text{STATES}(\pi) = \{C, A, D\}$ and $\text{CLOSURE}(\pi) = \{B\} \subseteq G$, thus, π is total. $\Diamond$

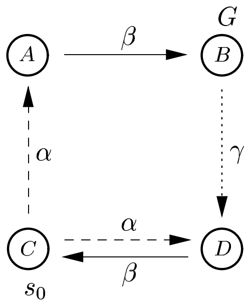


Figure 3.5: A non-deterministic planning problem with four states A, B, C, and D and three actions α (dashed), β (solid), and γ (dotted). The initial state is C while the set of goal states is a singleton set $\{B\}$.

Notice that the definition of a non-deterministic plan does not give any guarantees about goal achievement. The reason is that, in contrast to deterministic plans, it is natural to define a range of solutions classes. There currently exists three classes of non-deterministic plans called weak, strong cyclic, and strong [36, 37]. Following [42, 34], we use CTL to define these solutions. First, we need to define a Kripke structure to represent the execution behavior of a plan.

Definition 3.8 (Execution Model) *An execution model with respect to a non-deterministic plan π for the domain $\mathcal{D} = \langle S, Act, \rightarrow\rangle$ is a Kripke structure $\mathcal{M}(\pi) = \langle S, R\rangle$ where*

- $S = \text{CLOSURE}(\pi) \cup \text{STATES}(\pi) \cup G$,
- $\langle s, s'\rangle \in R$ *iff* $s \notin G,\ \exists a\,.\,\langle s, a\rangle \in \pi$ *and* $s \xrightarrow{a} s'$, *or* $s = s'$ *and* $s \in \text{CLOSURE}(\pi) \cup G$.

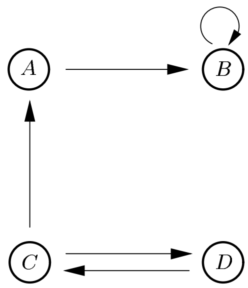


Figure 3.6: The execution model of the plan in Example 3.4.

Example 3.5 The execution model of the plan in Example 3.4 is shown in Figure 3.6. It has $S = \{A, B, C, D\}$ and $R = \{\langle A, B\rangle, \langle B, B\rangle, \langle C, A\rangle, \langle C, D\rangle, \langle D, C\rangle\}$. $\diamondsuit$

Notice that all execution paths are infinite which is required in order to define solutions in CTL. If a state is reached that is not covered by the plan (e.g., a goal state or a dead end), the postfix of the execution path from this states is an infinite repetition of it. Given a Kripke structure defining the execution of a plan, weak, strong cyclic, and strong plans are defined by the CTL formulas below.

Definition 3.9 (Weak, Strong Cyclic, and Strong Plans) *Given a non-deterministic planning problem $\mathcal{P} = \langle \mathcal{D}, s_0, G\rangle$ and a plan π for $\mathcal{D}$*

- *π is a weak solution iff $\mathcal{M}(\pi), s_0 \models \texttt{EF}\, G$,*
- *π is a strong cyclic solution iff $\mathcal{M}(\pi), s_0 \models \texttt{AGEF}\, G$,*
- *π is a strong solution iff $\mathcal{M}(\pi), s_0 \models \texttt{AF}\, G$.*

An execution of a strong plan is guaranteed to reach states covered by the plan until a goal state after a finite number of steps is reached. An execution of a strong cyclic plan is also guaranteed to reach states covered by the plan or a goal state. However, due to cycles, it may never reach a goal state. An execution of a weak plan may reach states not covered by the plan, it only guarantees that some execution exists that reaches the goal from the initial state.

3.2.1 Encoding NADL Domains

Compared with the wide range of deterministic planning languages, the number of non-deterministic planning languages is limited. The work presented in this thesis rests on the

Non-deterministic Agent Domain Language (NADL) [93] due to its explicit representation of environment actions.

NADL was developed as input language to the Universal Multi-agent Obbd-based Planner (UMOP) [93]. An NADL planning problem consists of: a set of *state variables*, a description of *system* and *environment agents*, and a specification of an *initial* and *goal condition*. The set of state variable assignments defines the state space of the domain. An agent's description is a set of *actions*. The agents change the state of the world by performing actions that are assumed to be executed synchronously and to have a fixed and equal duration. At each step, all of the agents perform exactly one action, and the resulting action tuple is a *joint action*. The system agents are assumed to be controllable, while the environment agents model the uncontrollable world. A valid domain description requires that the system and environment agents modify a disjoint set of state variables. Otherwise they may be able to control each other through their choice of actions. An action has three parts: a set of *modified* state variables, a *precondition* formula, and an *effect* formula. The next state value of the modified variables is defined by the effect formula and may depend on the value of the current state variables. During execution, the action has exclusive access to the modified state variables and it can not change the value of any other state variables. In order for the action to be applicable, the precondition formula must be satisfied in the current state. The values of state variables not modified by a joint action are unchanged. The initial and goal condition are formulas that must be satisfied in the initial state and the goal states, respectively. We assume that the initial condition only represents a single state.

Example 3.6 An NADL problem is shown in Figure 3.7. The problem has two state variables: a numerical one, position pos and a propositional one, $power$. The position is a natural number that can be represented by three Boolean variables. This gives pos the domain $\{0, 1, 2, 3, 4, 5, 6, 7\}$. The system is a robot moving between the eight positions. It has two actions *Right* and *Left*. The Right and Left actions have conditional effects described by an if-then-else operator ($\rightarrow$). If the power is on (that is, $power$ is $true$), they increase or decrease the position, otherwise they cause no position change. The Right action is non-deterministic. It may increase the position with either one or two. The Left action is deterministic. It always decreases the position with one. The environment is a human that controls the power with two actions *On* and *Off*. Since the system and environment must apply exactly one action at each step, there are four joint actions *Left-On*, *Left-Off*, *Right-On*, and *Right-Off*. Initially, the power is on and the robot is at position 0. $\diamondsuit$

There are two sources of non-determinism in NADL domains. The first is non-deterministic actions not constraining all their modified variables to a single value in the next state. The second is the uncontrollable actions of the environment. We define actions to be interfering if either

variables
  nat(3) pos
  bool $power$
system
    agt: Robot
      Right
        mod: pos
        pre: $pos < 6$
        eff: $power \rightarrow (pos' = pos + 1 \vee pos' = pos + 2), pos' = pos$
      Left
        mod: pos
        pre: $pos > 0$
        eff: $power \rightarrow pos' = pos - 1, pos' = pos$
environment
    agt: Human
      On
        mod: $power$
        pre: $\neg power$
        eff: $power'$
      Off
        mod: $power$
        pre: $power$
        eff: $\neg power'$
initially
  $pos = 0 \wedge power$
goal
  $pos = 7$

Figure 3.7: An NADL planning problem.

1. they have inconsistent effects, or
2. they constrain an overlapping set of state variables.

The first condition is due to the fact that state knowledge is expressed in a monotonic logic that cannot represent inconsistent knowledge. The second condition addresses the problem of sharing resources. We assume that each state variable at most can be accessed by a single action at each step even if the effect of several actions is consistent.

Abstract Syntax of NADL

The abstract syntax of an NADL description is a 7-tuple $D = \langle SV, S, E, Act, d, I, G\rangle$ where

- $SV = BVar \cup NVar$ is a finite set of state variables comprised of a finite set of Boolean variables, $BVar$, and a finite set of numerical variables with finite domains, $NVar$,
- S is a finite, nonempty set of system agents,
- E is a finite set of environment agents,
- Act is a set of action descriptions $\langle mod, pre, \mathit{eff}\rangle$ where mod is the set of state variables modified by the action, pre is a precondition state formula in the set $SForm$ and eff is an effect formula in the set $Form$. The sets $SForm$ and $Form$ are defined below.
- $d : Agt \to 2^{Act}$ is a function mapping agents ($Agt = S \cup E$) to their actions.
- $I \in SForm$ is the initial condition,
- $G \in SForm$ is the goal condition.

For a valid domain description, we require that actions of system agents modify a disjoint set of variables

$$\bigcup_{\substack{\alpha_s \in S \\ a \in d(\alpha_s)}} mod_a \cap \bigcup_{\substack{\alpha_e \in E \\ a \in d(\alpha_e)}} mod_a = \emptyset.$$

The set of formulas $Form$ is constructed from the following alphabet of symbols

- A finite set of current state v and next state v' variables, where $v, v' \in SV$,
- The natural numbers $\mathbb{N}$,
- The arithmetic operators $+$, $-$, $/$, and $*$,
- The relation operators $>$, $<$, $\leq$, $\geq$, $=$ and $\neq$,
- The Boolean operators $\neg$,$\vee$,$\wedge$,$\Rightarrow$,$\Leftrightarrow$ and $\to$,
- The special symbols $true$, $false$, parentheses and comma.

Arithmetic expressions are defined inductively by

- Every numerical state variable $v \in NVar$ is an arithmetic expression,
- A natural number is an arithmetic expression,
- If e_1 and e_2 are arithmetic expressions and $\oplus$ is an arithmetic operator, then $e_1 \oplus e_2$ is an arithmetic expression.

Finally, formulas $Form$ are defined inductively by

- $true$ and $false$ are formulas,
- Boolean state variables $v \in BVar$ are formulas,
- If e_1 and e_2 are arithmetic expressions and $\mathcal{R}$ is a relation operator, then $e_1 \; \mathcal{R} \; e_2$ is a formula,
- If f_1, f_2 and f_3 are formulas, so are $(\neg f_1)$, $(f_1 \vee f_2)$, $(f_1 \wedge f_2)$, $(f_1 \Rightarrow f_2)$, $(f_1 \Leftrightarrow f_2)$ and $(f_1 \rightarrow f_2, f_3)$.

Parentheses have their usual meaning and operators have their usual priority and associativity with the if-then-else operator "$\rightarrow$" given lowest priority. $SForm \subset Form$ is a subset of the formulas only referring to current state variables. All of the symbols in the alphabet of formulas have their usual meaning with the if-then-else operator $f_1 \rightarrow f_2, f_3$ being an abbreviation for $(f_1 \wedge f_2) \vee (\neg f_1 \wedge f_3)$.

The domain of a numerical state variable $v \in NVar$ is given by $dom(v) = \{0, 1, \cdots, t_v\}$, where $t_v > 0$. Let $JAct_s$ and $JAct_e$ denote the set of joint actions of system agents and environment agents, respectively

$$
\begin{aligned}
JAct_s &\equiv \prod_{\alpha_s \in S} d(\alpha_s), \\
JAct_e &\equiv \prod_{\alpha_e \in E} d(\alpha_e).
\end{aligned}
$$

Moreover, let $JAct$ denote the set of joint actions of system and environment agents $JAct \equiv JAct_s \times JAct_e$.

Encoding NADL Domains with BDDs

An NADL description $\langle SV, S, E, Act, d, I, G \rangle$ represents a non-deterministic planning problem $\mathcal{P}^{nd} = \langle \mathcal{D}^{nd}, s_0^{nd}, G^{nd} \rangle$ where $\mathcal{D}^{nd} = \langle S^{nd}, Act^{nd}, \rightarrow \rangle$ and

- $S^{nd} = \mathbb{B}^{BVar} \times \prod_{v \in NVar} dom(v)$,
- $Act^{nd} = JAct_s$,
- $s_0^{nd} \,:\, I(s_0^{nd})$,
- $G^{nd} = \{s \,:\, G(s)\}$,
- $s \xrightarrow{j_s} s'$ iff $R(s, j_s, s')$.

The transition relation is given by

$$R(s, j_s, s') = \exists j_e \in JAct_e \,.\, R(s, j, s')$$

where $j \in JAct$ is the joint action of the system and environment actions j_s and j_e given by $j = j_{s_1}, \cdots, j_{s_{|S|}}, j_{e_1}, \cdots, j_{e_{|E|}}$ and $R(s, j, s')$ is the transition relation of the joint system and environment actions. $R(s, j, s')$ is a conjunction of three relations A, F, and I

$$R(s, j, s') = A(s, j, s') \wedge F(s, j, s') \wedge I(j).$$

A defines the constraints on the current state s and next state s' caused by the actions in the joint action j. A further ensures that actions with inconsistent effects cannot be applied concurrently, since A reduces to false if any pair of actions in j has inconsistent effects. Thus, A also ensures the first condition for avoiding interference between concurrent actions. We have

$$A(s, j, s') = \bigwedge_{i=1}^{|Agt|} \Big(pre_{j_i}(s) \wedge \mathit{eff}_{j_i}(s, s') \Big).$$

F is a frame relation ensuring that unmodified variables are unchanged

$$F(s, j, s') = \bigwedge_{v \in SV \setminus C} (v = v')$$

where $C = \bigcup_{i=1}^{|Agt|} mod_{j_i}$.

Finally, I ensures the second condition for avoiding interference between concurrent actions.

$$I(j) = \bigwedge_{i \neq k} \Big(mod_{j_i} \cap mod_{j_k} = \emptyset \Big).$$

An NADL domain corresponds to a synchronous system where each agent is an activity. From the discussion in Section 2.3.1, we can therefore expect to be able to use a conjunctive partitioning to represent the transition relation of the domain. The definition of

$R(s, j, s')$ above verifies this, since $R(s, j, s')$ is a conjunction of subexpressions. However, the existential quantification in the expression

$$R(s, j_s, s') = \exists j_e \in JAct_e \,.\, R(s, j, s').$$

does not distribute over the conjunction of subexpressions in $R(s, j, s')$. It is possible though, to use the early quantification technique explained in Section 2.3.1 to obtain a conjunctive partitioning of $R(s, j_s, s')$ by moving subexpression out of scope of the existential quantification.

Notice that, for non-deterministic planning, the BDD encoding of the transition relation needs to be complete in the sense that the action of a transition must be encoded in the BDD. For deterministic planning, the disjunctive partitioning of the transition relation could be used to represent actions. This is not possible for non-deterministic planning since the search algorithms reason about state-action pairs (SAs) and synthesize plans represented by a set of SAs.

In the remainder of the thesis, we focus on simple multi-agent problems with a single system agent and at most one environment agent. We have developed a specialized version of NADL called NADL$^+$ where the system and environment is represented by a set of actions instead of a set of agents. In addition, NADL$^+$ has features to support guided BDD-based search and failure effects of actions. If no environment actions exist, it is straight forward to encode the transition relation of an NADL$^+$ domain as a disjunctive partitioning. Otherwise, if environment actions exist, we “flatten” the action descriptions by computing each joint action, which again makes it possible to represent the transition relation as a disjunctive partitioning. This can be done efficiently due to the relatively small number of joint-actions. NADL$^+$ is described in more detail in Appendix A.

3.2.2 Planning Algorithms

Weak, strong cyclic, and strong plans can be synthesized by a backward breadth-first search from the goal states to the initial states. The search algorithm is shown in Figure 3.8. In each iteration (l.2-7), a precomponent P_c of the plan is computed from the states C currently covered by the plan. If the precomponent is empty, a fixed point of P has been reached that does not cover the initial states and “no solution exists” is returned. Otherwise, the precomponent is added to the plan and the states in the precomponent are added to the set of covered states (l.6-7). The precomponent function must fulfill the specification given below.

Definition 3.10 (Precomponent Function) *A valid precomponent function* $\textsc{PreComp}(C) : 2^S \rightarrow 2^{S \times Act}$ *must terminate. In addition, For any state-action pair in the precomponent*

function NDP(s_0, G)
$P \leftarrow \emptyset; C \leftarrow G$
while $s_0 \notin C$
  $P_c \leftarrow$ PRECOMP(C)
  if $P_c = \emptyset$ **then return** "no solution exists"
  else
    $P \leftarrow P \cup P_c$
    $C \leftarrow C \cup$ STATES(P_c)
return P

Figure 3.8: A generic algorithm for synthesizing non-deterministic plans.

$\langle s, a \rangle \in$ PRECOMP(C)*, we have* $s \notin C$.

Since the set of states is finite and the precomponent function terminates, P must eventually reach a maximum size. Thus, it can be shown that NDP terminates.

Theorem 3.1 (Termination) *NDP terminates.*

Proof. Given in Appendix B □

The Strong, strong cyclic, and weak planning algorithms only differ by the definition of the precomponent. The core operation is to find the preimage where states are associated with actions

$$\text{PREIMGSA}(C) \equiv \exists \vec{v}\,' \,.\, R(\vec{v}, \vec{a}, \vec{v}\,') \wedge C(\vec{v})[\vec{v}/\vec{v}\,']. \tag{3.7}$$

As a set computation PREIMGSA(C) is defined by

$$\text{PREIMGSA}(C) \equiv \{\langle s, a \rangle \,:\, \text{NEXT}(s, a) \cap C \neq \emptyset\}. \tag{3.8}$$

The weak and strong precomponent is the set of SAs given by

$$\text{PRECOMPW}(C) \equiv \text{PREIMGSA}(C) \setminus C \times Act \tag{3.9}$$
$$\text{PRECOMPS}(C) \equiv (\text{PREIMGSA}(C) \setminus \text{PREIMGSA}(\overline{C})) \setminus C \times Act \tag{3.10}$$

The strong cyclic precomponent PRECOMPSC(C) can be generated by iteratively extending a set of candidate SAs (wSA) and pruning it until a fixed point is reached [34]. The precomponent function is shown in Figure 3.9.

Let WEAK, STRONGCYCLIC, and STRONG denote the NDP algorithm using PRECOMPW, PRECOMPSC, and PRECOMPS, respectively. It is shown in Appendix B that

function PRECOMPSC(C)
$wSA \leftarrow \emptyset$
repeat
  $OldwSA \leftarrow wSA$
  $wSA \leftarrow \text{PREIMGSA}(\text{STATES}(wSA) \cup C) \setminus C \times Act$
  $scSA \leftarrow \text{SCPLANAUX}(wSA, C)$
until $scSA \neq \emptyset \ \vee \ wSA = OldwSA$
return $scSA$

function SCPLANAUX$(startSA, C)$
$SA \leftarrow startSA$
repeat
  $OldSA \leftarrow SA$
  $SA \leftarrow \text{PRUNEOUTGOING}(SA, C)$
  $SA \leftarrow \text{PRUNEUNCONNECTED}(SA, C)$
until $SA = OldSA$
return SA

function PRUNEOUTGOING(SA, C)
$NewSA \leftarrow SA \setminus \text{PREIMGSA}(\overline{C \cup \text{STATES}(SA)})$
return $NewSA$

function PRUNEUNCONNECTED(SA, C)
$NewSA \leftarrow \emptyset$
repeat
  $OldSA \leftarrow NewSA$
  $NewSA \leftarrow SA \cap \text{PREIMGSA}(C \cup \text{STATES}(NewSA))$
until $NewSA = OldSA$
return $NewSA$

Figure 3.9: The strong cyclic precomponent function.

WEAK, STRONGCYCLIC, and STRONG are *sound* and *complete* and have valid precomponents. Since we have shown the generic non-deterministic algorithm then terminates, we have

Theorem 3.2 (Correctness of Weak, StrongCyclic, and Strong) *The* WEAK*,* STRONGCYCLIC*, and* STRONG *planning algorithms are correct. The algorithms return "no solu-*

tion exists" iff no solution exists, otherwise they return a valid solution.

Proof. This follows from the soundness, completeness, and termination theorems of each algorithm proven in Appendix B. □

Due to the breadth-first search carried out by the non-deterministic planning algorithm, weak solutions have minimum length best-case execution paths and strong solutions have minimum length worst-case execution paths [34]. Formally, for a non-deterministic planning domain $\mathcal{D}$ and a plan π of $\mathcal{D}$ let

$$\text{EXEC}(s, \pi) \equiv \{q \,:\, q \text{ is a path of } \mathcal{M}(\pi) \text{ and } q_0 = s\} \tag{3.11}$$

denote the set of execution paths of π starting at s. Let the length of a path $q = q_0 q_1 \cdots$ with respect to a set of states C be defined by

$$|q|_C \equiv \begin{cases} i & : \text{ if } q_i \in C \text{ and } q_j \notin C \text{ for } 0 \leq j < i \\ \infty & : \text{ otherwise.} \end{cases} \tag{3.12}$$

We will say that an execution path q *reaches* a state s iff $|q|_{\{s\}} \neq \infty$. In addition, we will call a state s *connected* to a set of states C by a plan π iff $\mathcal{M}(\pi), s \models \mathsf{EF}\, C$. Let $\text{MIN}(s, C, \pi)$ and $\text{MAX}(s, C, \pi)$ denote the minimum and maximum length of an execution path from s to C of a plan π

$$\text{MIN}(s, C, \pi) \equiv \min_{q \in \text{EXEC}(s,\pi)} |q|_C \tag{3.13}$$

$$\text{MAX}(s, C, \pi) \equiv \max_{q \in \text{EXEC}(s,\pi)} |q|_C. \tag{3.14}$$

Similarly, let Π denote the set of all plans of $\mathcal{D}$ and let $\text{WDIST}(s, C)$ (weak distance) and $\text{SDIST}(s, C)$ (strong distance) denote the minimum of $\text{MIN}(s, C, \pi)$ and $\text{MAX}(s, C, \pi)$ for any plan $\pi \in \Pi$ of $\mathcal{D}$

$$\text{WDIST}(s, C) \equiv \min_{\pi \in \Pi} \text{MIN}(s, C, \pi) \tag{3.15}$$

$$\text{SDIST}(s, C) \equiv \min_{\pi \in \Pi} \text{MAX}(s, C, \pi). \tag{3.16}$$

It can be shown that the WEAK and STRONG algorithms are optimal with respect to weak and strong distance.

Theorem 3.3 (Optimality of Weak and Strong)

- *If π is a solution returned by* $\text{WEAK}(s_0, G)$ *then* $\text{MIN}(s_0, G, \pi) = \text{WDIST}(s_0, G)$.
- *If π is a solution returned by* $\text{STRONG}(s_0, G)$ *then* $\text{MAX}(s_0, G, \pi) = \text{SDIST}(s_0, G)$.

Proof. Follows from the optimality proofs of WEAK and STRONG given in Appendix B. □

A limitation of strong cyclic and strong planning compared to weak planning is that strong cyclic and strong plans often do not exist because it is impossible to avoid dead ends. Consider generating a non-deterministic plan for a system that can be in a set of bad states, a set of good states or a set irrecoverable failed states (dead-ends). Assume that

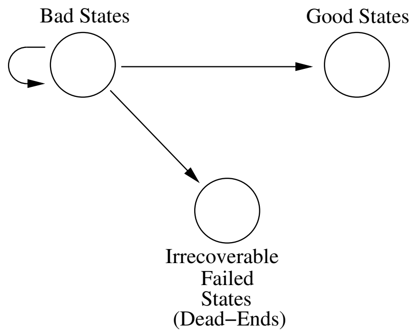


Figure 3.10: System with irrecoverable states.

there exist actions that can bring the system from any bad state to a good state. However, these actions may fail and cause transitions to bad states or even irrecoverable failed states (see Figure 3.10). No strong nor strong cyclic plan can be found since an irrecoverable state can be reached from any initial state. There only exists a weak plan for this problem. However, weak plans are mostly useless since actions are chosen without reasoning about their worst-case behavior.

Another limitation of strong and strong cyclic plans is their inherent pessimism. Consider for example the domain illustrated in Figure 3.11. The domain consists of $n+1$ states and two different actions (dashed and solid). The only strong cyclic and strong solution is

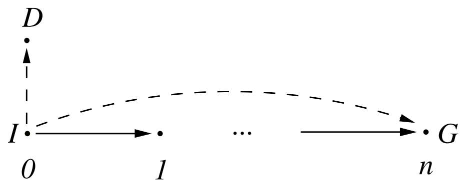


Figure 3.11: A domain with two actions (drawn as solid and dashed arrows) illustrating the possible loss of short execution paths. I and G are the initial and goal state, respectively.

$\{\langle 0, solid\rangle, \langle 1, solid\rangle, \cdots, \langle n-1, solid\rangle\}$. There is a single execution path associated with this plan that reaches the goal state in n steps. However, a weak plan $\{\langle 0, dashed\rangle\}$ may be preferable since the probability of its best-case execution length of 1 may be much higher than its worst-case infinite execution length.

Chapter 4

State-Set Branching

In this chapter, we introduce a new framework called *state-set branching* [88, 89, 90, 91]. State-set branching combines BDD-based search and heuristic search. The philosophy of state-set branching is that the information represented by BDDs must be semantically closely related in order for the BDD operations to work efficiently. Hence, we separate the representation of information used to guide the search algorithm from the representation of states and transitions and only use BDDs to encode the latter. The framework has two independent parts: a modification of the Best-First Search algorithm (BFS) described in Section 2.4 to a new algorithm called *Best-Set-First Search* (BSFS), and an efficient BDD-based implementation of this algorithm based on a partitioning of the transition relation called *branching partitioning*. In Section 4.1, we introduce the BSFS algorithm and show that it applies to any classical BFS algorithm, any transition cost function, heuristic function, and node-evaluation function. In Section 4.2, we define branching partitioning and describe how this new BDD partitioning technique can be used to implement the BSFS algorithm. Finally, Section 4.3 describes an experimental evaluation of two implementations of A* called GHSETA* and FSETA*. The performance of these algorithms is compared to unguided BDD-based search, ordinary single-state A*, and BDDA*, the only previous BDD-based implementation of A*. The evaluation includes 8 search domains ranging from VLSI-design with synchronous actions, to classical AI problems such as $(n^2 - 1)$-Puzzles, Blocks World and problems used in the AIPS 1998, 2000 and 2002 planning competitions [113, 4, 115]. We apply four different families of heuristic functions ranging from the minimum Hamming distance to the sum of Manhattan distances for the $(n^2 - 1)$-Puzzle, and HSPr [20] for planning problems. The experimental evaluation shows that GHSETA* and FSETA* consistently outperform single-state A*, except when the heuristic is very strong. In addition, we show that it can improve the complexity of single-state search exponentially and that it often dominates both single-state A* and blind BDD-based search by several orders of magnitude. Moreover, it consistently outperforms BDDA*.

4.1 Best-Set-First Search

The Best-Set-First Search (BSFS) algorithm generalizes BFS to build a search tree where each search node contains a set of states associated with the same search information. There are two main properties of BDD-based search techniques that this algorithm exhibits.

1. It can exploit the ability of the image computation to find next states of a set of states effectively when expanding a search node.
2. Since a search node contains states associated with the same search information, it can avoid using inefficient symbolic arithmetic operations to find the search information associated with states in child nodes.

BSFS can implement heuristic search tree algorithms where the search node information used to prioritize the node expansion can be computed by associating each transition $\langle s, a, s' \rangle$ of the search domain with a change $\delta I(s, a, s')$ of the search node information. Thus, if s belongs to a node with information I and s' is reached with transition $\langle s, a, s' \rangle$ then s' belongs to a search node with information $I + \delta I(s, a, s')$. For A*, the search node information can be one or two dimensional: either it is the f-value or the g and h-value. In the first case, $\delta I(s, a, s')$ is the change in f-value caused by the transition.

Example 4.1 The δh, δg, and resulting δf values of the problem introduced in Example 2.8 are shown in Figure 4.1. ♢

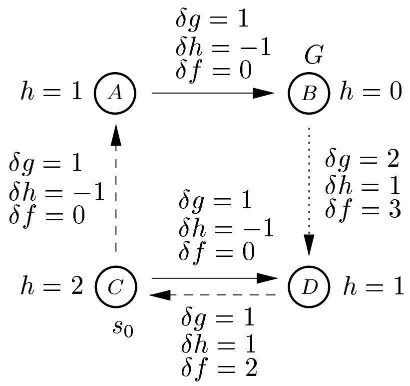


Figure 4.1: The δf-values of the search problem introduced in Example 2.8.

The BSFS algorithm shown in Figure 4.2 is almost identical to the ordinary BFS algorithm defined in Figure 2.8. However, the state-set version builds a search tree during the search process where each search node contains a set of states. Multiple states in each node emerge because child nodes with identical node information are coalesced by the

function BSFS(s_0, I_0, G)
1 *frontier* $\leftarrow$ MAKEQUEUE$(\langle\{s_0\}, I_0\rangle)$
2 **loop**
3   **if** $|\textit{frontier}| = 0$ **then return** ”no solution exists”
4   $\langle S, I\rangle \leftarrow$ REMOVETOP$(\textit{frontier})$
5   **if** $S \cap G \neq \emptyset$ **then return** EXTRACTSOLUTION$(\textit{frontier}, \langle S \cap G, I\rangle)$
6   *frontier* $\leftarrow$ ENQUEUEANDMERGE$(\textit{frontier},$ STATESETEXPAND$(\langle S, I\rangle))$

Figure 4.2: The Best-Set-First Search (BSFS) algorithm.

STATESETEXPAND function in line 6 and because the ENQUEUEANDMERGE function may merge child nodes with nodes on the *frontier* queue having identical node information. The STATESETEXPAND function is defined in Figure 4.3.

function STATESETEXPAND$(\langle S, I\rangle)$
1 **child** $\leftarrow$ *emptyMap*
2 **foreach** state s **in** S
3   **foreach** transition $\langle s, a, s'\rangle$
4     $I_c \leftarrow I + \delta I(s, a, s')$
5   **child**$[I_c] \leftarrow$ **child**$[I_c] \cup \{s'\}$
6 **return** MAKENODES(**child**)

Figure 4.3: The STATESETEXPAND function.

Child states with node information I are stored in $child[I]$. The outgoing transitions from each state in the parent node are used to find all successor states. The function MAKENODES called at line 6 constructs the child nodes from the completed child map. Each child node contains states with the same search information. However, there may exist several nodes with the same node information. In addition, MAKENODES may prune some of the child states (e.g., to implement cycle detection in A*).

Example 4.2 Figure 4.4 shows the search tree traversed by the BSFS algorithm for A* applied to the problem in Example 4.1. $\diamondsuit$

In order to introduce multiple states in each search node and reduce the number of search nodes, the ENQUEUEANDMERGE function of the BSFS algorithm may merge nodes on the search frontier having identical search information. This, however, transforms the search tree into a Directed Acyclic Graph (DAG). We will refer to this DAG as a *search structure*.

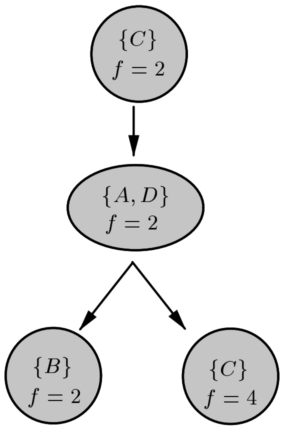


Figure 4.4: State-set search tree example.

Lemma 4.1 *The search structure build by the BSFS algorithm is a DAG where every node $\langle S', I'\rangle$ different from a root node $\langle \{s_0\}, I_0\rangle$ has a set of predecessor nodes. For each state $s' \in S'$ in such a node there exists an action a and a predecessor $\langle S, I\rangle$ with a state $s \in S$ such that $s \xrightarrow{a} s'$ and $I' = I + \delta I(s, a, s')$.*

Proof. By induction on the number of loop iterations, we get that the search structure after the first iteration is a DAG consisting of a root node $\langle \{s_0\}, I_0\rangle$. For the inductive step, assume that the search structure is a DAG with the desired properties after n iterations of the loop (see Figure 4.2). If the algorithm in the next iteration terminates in line 3 or 5, the search structure is unchanged and therefore a DAG with the required format. Assume that the algorithm does not terminate and that $\langle S, I\rangle$ is the node removed from the top of $frontier$. The node is expanded by forming child nodes with the STATESETEXPAND function in line 6. According to the definition of this function, for any state $s' \in S'$ in a child node $\langle S', I'\rangle$ there exists an action a and some state $s \in S$ in $\langle S, I\rangle$ such that $s \xrightarrow{a} s'$ and $I' = I + \delta I(s, a, s')$. Thus $\langle S, I\rangle$ is a valid predecessor for all states in the child nodes. Furthermore, since all child nodes are new nodes, no cycles are created in the search structure which therefore remains a DAG. If a child node is merged with an old node when enqueued on $frontier$ the resulting search structure is still a DAG because all nodes on $frontier$ are unexpanded and therefore have no successor nodes that can cause cycles. In addition, each state in the resulting node obviously has the required predecessor nodes. $\square$

Lemma 4.2 *For each state $s' \in S'$ of a node $\langle S', I'\rangle$ in a finite search structure of the BSFS algorithm there exists a path $q_0 \cdots q_n$ with associated actions $\pi = a_1 \cdots a_n$ in the search domain such that $q_n = s'$ and $I' = I_0 + \sum_{i=1}^{n} \delta I(q_{i-1}, a_i, q_i)$.*

Proof. We will construct the path by tracing the edges backwards in the search structure. Let $p_n = s'$. According to Lemma 4.1 there exists a predecessor $\langle S, I \rangle$ to $\langle S', I' \rangle$ such that for some state $p_{n-1} \in S$ and action $\alpha_n \in Act$ we have $p_{n-1} \xrightarrow{\alpha_n} p_n$ and $I' = I + \delta I(p_{n_1}, \alpha_n, p_n)$. Continuing the backward traversal from p_{n-1} must eventually terminate since the search structure is finite and acyclic. Moreover, the traversal will terminate at the root node because this is the only node without predecessors. Assume that the backward traversal terminates after n iterations. Then $q_0 \cdots q_n = p_0 \cdots p_n$ and $\pi = \alpha_1 \cdots \alpha_n$ $\Box$

The EXTRACTSOLUTION function in line 5 of the BSFS algorithm uses the backward traversal described in the proof of Lemma 4.2 to extract a solution. We can now prove soundness of the BSFS algorithm.

Theorem 4.1 (Soundness of BSFS) *If the BSFS algorithm returns a solution $\pi = a_1 \cdots a_n$ with associated path $q_0 \cdots q_n$ and the search node information of q_n's search node is I then π is a valid solution and $I = I_0 + \sum_{i=1}^{n} \delta I(q_{i-1}, a_i, q_i)$.*

Proof. Since $q_n \in G$ it follows from Lemma 4.2 and the definition of EXTRACTSOLUTION that π is a solution to the search problem and $I = I_0 + \sum_{i=1}^{n} \delta I(q_{i-1}, a_i, q_i)$. $\Box$

It is not possible to show that the BSFS algorithm in general is complete since it covers incomplete algorithms such as pure heuristic search. However, it follows from the optimality proofs below that BSFS is complete when implementing the A* algorithm.

Example Implementations

The BSFS algorithm can be used to implement all classical variants of the BFS algorithm including pure heuristic search, A*, weighted A*, uniform cost search, beam search, and hill climbing. With some modifications, it also covers iterative deepening heuristic search algorithms such as IDA*.

Pure heuristic search is implemented by using the values of the heuristic function as search node information and sorting the nodes on the frontier in ascending order such that the top node contains states with least h-value. The search node information of the initial state is $I_0 = h(s_0)$ and each transition $\langle s, a, s' \rangle$ is associated with the change in h, that is, $\delta I(s, a, s') = h(s') - h(s)$. In each iteration, this pure heuristic search algorithm will expand all states with least h-value on the frontier given that all nodes with identical h-value are merged on the frontier queue.

A* can be implemented by setting $I_0 = h(s_0)$ and $\delta I(s, a, s') = c(a) + h(s') - h(s)$. In this way, the search node information equals the f-value of the states belonging to the nodes. Again, nodes on the frontier are sorted ascendingly. We call a particular implementation of this algorithm where all nodes with identical f-value on the frontier queue are merged for FSETA*. An A* implementation with cycle detection must keep track of g and h separately and prune child states reached previously with a lower g-value. Thus, $I_0 = (0, h(s_0))$ and $\delta I(s, a, s') = (c(a), h(s') - h(s))$. The frontier is, as usual, sorted according with respect to the evaluation function $f(n) = g(n) + h(n)$. The resulting algorithm is called GHSETA*. Compared to FSETA*, GHSETA* does not merge nodes that have identical f-value but different g and h-values. In each iteration, it may therefore only expand a subset of the states on the frontier with minimum f-value. A number of other improvements have been integrated in GHSETA*. First, it uses a tie breaking rule for nodes with identical f-value that chooses the node with the least h-value. Thus, in situations where all nodes on the frontier have $f(n) = C^*$, the algorithm focuses the search in a DFS fashion. The reason is that a node at depth level d in this situation must have greater h-value than a node at level $d + 1$ due to the non-negative transition costs. In addition, it merges two nodes on the frontier only if the space used by the resulting node is less than an upper-bound u. This may help to focus the search further in situations where the space requirements of the frontier nodes grow fast with the search depth. Both GHSETA* and FSETA* can easily be extended to the weighted A* algorithms described in Section 2.4. Using an approach similar to Pearl [130], FSETA* and GHSETA* can be shown to be optimal given an admissible heuristic. In particular this is true when using the trivial admissible heuristic function $h(n) = 0$ of uniform cost search.

Lemma 4.3 *Assume* FSETA* *and* GHSETA* *apply an admissible heuristic and $q_0 \cdots q_n$ is the path associated with an optimal solution $\pi = a_1 \cdots a_n$, then at any time before* FSETA* *and* GHSETA* *terminate there exists a frontier node $\langle S, I \rangle$ with $q_i \in S$ such that $I \leq C^*$ and $q_0 \cdots q_i$ is the search path associated with q_i.*

Proof. A node $\langle S, I \rangle$ containing q_i with associated search path $q_0 \cdots q_i$ must be on the frontier since a node containing s_0 was initially inserted on the frontier and FSETA* and GHSETA* terminate if a node containing the goal state s_n is removed from the frontier. We have $I = cost(a_1 \cdots a_i) + h(q_i)$. The path $q_0 \cdots q_i$ is a prefix of an optimal solution, thus $cost(a_1 \cdots a_i)$ must be the minimum cost of reaching q_i. Since the heuristic function is admissible, we have $h(q_i) \leq h^*(q_i)$ which gives $I \leq C^*$. □

Theorem 4.2 (Optimality of fSetA* and ghSetA*) *Given an admissible heuristic function,* FSETA* *and* GHSETA* *are optimal.*

Proof. Suppose FSETA* or GHSETA* terminates with a solution derived from a frontier node with $I > C^*$. Since the node was at the top of the frontier queue, we have

$$I < f(n) \ \forall n \in frontier.$$

Thus, prior to termination, all nodes on the frontier satisfied $f(n) > C^*$. However, this contradicts Lemma 4.3 that states that any optimal path has a node on the frontier any time prior to termination with $I \leq C^*$. $\square$

IDA* performs a depth-first search in the search tree bounded by a limit f_{limit} on the f-values of search nodes. Initially, f_{limit} is equal to the f-value of the initial state. In each iteration, f_{limit} is increased by the minimum value that the previous search exceeded f_{limit} by. A similar algorithm can be defined for a state-set search structure where child nodes with identical f-values are combined.

4.2 BDD-Based Implementation

The motivation for defining the BSFS algorithm is that it can be efficiently implemented with BDDs. In this section, we define a new BDD technique called *branching partitioning* to effectively expand search nodes where the sets of states are represented by BDDs.

The BDD-based BSFS algorithm represents the states in each search node by a BDD. This may lead to exponential space savings compared to the explicit state representation used by ordinary implementations of best-first search. However, if we want exponential space savings to translate into an exponential time savings, we also need an implicit approach for computing the expand operation. The image computation can be applied to find all next states of a set of states implicitly, but we need a way to partition the next states into child nodes with identical node information. The expand operation could be carried out in two phases, where the first finds all the next states using the image computation, and the second splits this set of states into child nodes [179]. A more efficient approach, however, is to split up the image computation such that the second phase is integrated in the first phase without a significant computational overhead. We call this branching partitioning.

4.2.1 Disjunctive Branching Partitioning

For disjunctive partitioning the approach is straight-forward. We simply ensure that each partition contains transitions with the same search information change. The result is called a *disjunctive branching partitioning*.

Definition 4.1 (Disjunctive Branching Partitioning) *A disjunctive branching partitioning is a disjunctive partitioning $R_1(\vec{x}_1, \vec{y}'_1), \cdots, R_n(\vec{x}_n, \vec{y}'_n)$ where each subrelation represents a set of transitions with the same search node information change.*

Notice, that there may exist several partitions with identical information change. In practice, it is often more efficient to merge some of these partitions even though more variables will be modified by the resulting partitions.

So far, an unresolved problem is how to find the search node information change of each transition efficiently. It is intractable to compute $h(s)$ explicitly for each state since the number of states grows exponentially with the number of state variables of the domain. In practice, however, it turns out that δh of an action often is independent of which state it is applied in. This is not a coincidence. Heuristics are relaxations that typically are based on ignoring interactions between actions in the domain. Thus, the effect of an action can often be associated with a particular δh value. In the worst case, it may be necessary to encode the heuristic function symbolically with a BDD $h(\vec{e}, \vec{v})$ where the vector of Boolean variables $\vec{e}$ encodes the heuristic value in binary of the state represented by $\vec{v}$. We can then compute $\delta h(s, s')$ symbolically with

$$\delta h(\vec{v}, \vec{v}\,', \vec{d}) \equiv h(\vec{e}, \vec{v}) \wedge h(\vec{e}\,', \vec{v}\,') \wedge \vec{d} = \vec{e}\,' - \vec{e} \tag{4.1}$$

where $\vec{d}$ encodes the value of $\delta h(s, s')$ in binary. This computation avoids iterating over all states. In addition, it only needs to be carried out once prior to search. For all of the heuristics studied in this thesis (including several classical heuristics), it has not been necessary to perform this symbolic computation. Instead, the δh value of each action has been independent or close to independent of the state the action is applied in.

Example 4.3 For the search problem in Example 4.1, we get at least three subrelations corresponding to the three distinct δf-values

$$\begin{array}{lclcll}
\delta f_1 & = & 0 & & & \\
R_1(\vec{v}, \vec{v}\,') & = & \neg v_1 \wedge \neg v_2 & \wedge & v_1' \wedge \neg v_2' & \vee \\
 & & \neg v_1 \wedge v_2 & \wedge & \neg v_1' \wedge \neg v_2' & \vee \\
 & & \neg v_1 \wedge v_2 & \wedge & v_1' \wedge v_2' & \\
 & & & & & \\
\delta f_2 & = & 2 & & & \\
R_2(\vec{v}, \vec{v}\,') & = & v_1 \wedge v_2 & \wedge & \neg v_1' \wedge v_2' & \\
 & & & & & \\
\delta f_3 & = & 3 & & & \\
R_3(\vec{v}, \vec{v}\,') & = & v_1 \wedge \neg v_2 & \wedge & v_1' \wedge v_2'. &
\end{array}$$

$\Diamond$

Assume that the search node information change associated with subrelation i is δI_i and that there are n subrelations. Let $\text{IMG}_i(C)$ denote the image of the transitions in subrelation i

$$\textsc{Img}_i(C) \quad \equiv \quad \Big(\exists \vec{y}_i \,.\, C(\vec{v}) \wedge R_i(\vec{x}_i, \vec{y}\,'_i)\Big)[\vec{y}\,'_i/\vec{y}_i]. \tag{4.2}$$

The StateSetExpand function in Figure 4.3 can then be implemented with BDDs as shown in Figure 4.5.

function DisjunctiveStateSetExpand$(\langle S, I\rangle)$
child $\leftarrow$ *emptyMap*
for $i = 1$ **to** n
  $I_c \leftarrow I + \delta I_i$
  child$[I_c] \leftarrow$ **child**$[I_c] \cup \textsc{Img}_i(S)$
return MakeNodes(**child**)

Figure 4.5: The StateSetExpand function for a disjunctive branching partitioning.

4.2.2 Conjunctive Branching Partitioning

An efficient implicit node expansion computation is also possible to define for a conjunctive partitioning. Consider the synchronous composition of the n subsystems in Figure 2.6. Assume that the search node information change of a joint activity equals the sum of information changes of each activity. We can then represent a conjunctive branching partitioning as n disjunctive branching partitionings where each disjunctive branching partitioning represents the subrelations of the activities.

Definition 4.2 (Conjunctive Branching Partitioning) *A conjunctive branching partitioning $P_1, \cdots, P_n$ is a set of disjunctive branching partitionings*

$$P_i = R_i^1(\vec{x}_i, \vec{y}\,'_i), \cdots, R_i^{r_i}(\vec{x}_i, \vec{y}\,'_i)$$

for $1 \leq i \leq n$.

Since the subsystems are synchronous, we require that the sets of variables in $\vec{y}\,'_1, \cdots, \vec{y}\,'_n$ form a partitioning of the state variables $\vec{v}\,'$. Assume that the search node information change of $R_i^j(\vec{x}_i, \vec{y}\,'_i)$ is δI_i^j. Further let

$$\textsc{SubComp}_i^j(\phi) \quad \equiv \quad \exists \vec{z}_i \,.\, \phi(\vec{v}, \vec{v}\,') \wedge R_i^j(\vec{x}_i, \vec{y}\,'_i) \tag{4.3}$$

where ϕ represents an intermediate computation result. As for an ordinary conjunctive image computation, we require $Z_j \cap \bigcup_{i=j+1}^{n} X_i = \emptyset$ for $1 \leq j < n$ and $\bigcup_{i=1}^{n} Z_i = \{v_1, \cdots, v_n\}$.

The conjunctive state-set expansion function is then defined as shown in Figure 4.6. The

function CONJUNCTIVESTATESETEXPAND($\langle S, I \rangle$)
child $\leftarrow$ *emptyMap*
child$[I] \leftarrow S$
for $i = 1$ **to** n
  newChild $\leftarrow$ *emptyMap*
  foreach entry $\langle \phi, \delta I \rangle$ **in child**
    for $j = 1$ **to** r_i
      $I_c \leftarrow \delta I + \delta I_i^j$
      newChild$[I_c] \leftarrow$ **newChild**$[I_c] \vee$ SUBCOMP$_i^j(\phi)$
  child $\leftarrow$ **newChild**
return MAKENODES(**child**)

Figure 4.6: The STATESETEXPAND function for a conjunctive branching partitioning.

outer loop of the conjunctive state-set expansion function performs n iterations. In iteration i, the next value of the variables $\vec{y}_i$ is computed. In the end, the map *child* contains sets of next states with identical search node information.[1] In the worst case, the number of child nodes will grow exponentially with the number of activities. However, in practice this blow-up of child nodes may be avoided due to the merging of nodes with identical search node information during the computation.

4.3 Experimental Evaluation

Even though state-set branching applies to weighted A* and pure heuristic search, the experimental evaluation focuses on evaluating the two implementations FSETA* and GHSETA* of the A* algorithm. There are several reasons for this. First, we are interested in finding optimal or near optimal solutions, and for pure heuristic search, the whole emphasis would be on the quality of the heuristic function rather than the efficiency of the search approach. Second, the behavior of A* has been extensively studied, and finally, we

[1]The function MAKENODES generates search nodes from the map. In addition, it substitutes the variables of the BDDs encoding next states from primed to unprimed state variables.

wish to compare with the BDDA* algorithm. Readers interested in the performance of state-set branching algorithms of weighted A* with weight settings other than $w = 0.5$ (see Equation 2.17) are referred to [89].

All experiments have been carried out with the BIFROST 0.7 search engine using the experimental setting described in Appendix A. The input to BIFROST is a search problem defined in the STRIPS part of PDDL or NADL$^+$ described in Appendix A where action costs and heuristics can be defined. The performance of 6 algorithms GHSETA*, FSETA*, BIDIR, BDDA*, and iBDDA* is investigated. The GHSETA*, FSETA*, and BIDIR search algorithms have been described in Section 4.1 and Section 3.1.2. The A* algorithm manipulates and represents states explicitly. Due to the different state representations, specialized versions have been made for the $(n^2 - 1)$-Puzzles, the DVM domain, and the FG^k domain described below. In addition, a general version for PDDL planning is implemented in BIFROST 0.7 and represents states as sets of facts and actions in the usual STRIPS fashion. All of the single-state A* algorithms are implemented with cycle detection. The BDDA* algorithm has been implemented in BIFROST 0.7 as described in [53]. It is shown in Figure 4.7. BDDA* can solve search problems only in domains

function BDDA*(s_0, G)
1 $\quad open(\vec{f},\vec{v}) \leftarrow h(\vec{f},\vec{v}) \wedge s_0(\vec{v})$
2 $\quad$ **while** $(open \neq \emptyset)$
3 $\quad\quad (f_{min}, min(\vec{v}), open'(\vec{f},\vec{v})) \leftarrow$ GOLEFT$(open)$
4 $\quad\quad$ **if** $\exists \vec{v} \,.\, min(\vec{v}) \wedge G(\vec{v})$ **return** f_{min}
5 $\quad\quad open''(\vec{f}',\vec{v}') \leftarrow \exists \vec{v} \,.\, min(\vec{v}) \wedge T(\vec{v},\vec{v}') \wedge$
6 $\quad\quad\quad \exists \vec{e} \,.\, h(\vec{e},\vec{v}) \wedge \exists \vec{e}' \,.\, h(\vec{e}',\vec{v}') \wedge (\vec{f}' = f_{min} + \vec{e}' - \vec{e} + 1)$
7 $\quad\quad open(\vec{f},\vec{v}) \leftarrow open'(\vec{f},\vec{v}) \vee open''(\vec{f}',\vec{v}')[\vec{f}' \setminus \vec{f}, \vec{v}' \setminus \vec{v}]$

Figure 4.7: The BDDA* algorithm.

with unit transition costs. The search frontier is represented by a single BDD $open(\vec{f},\vec{v})$. This BDD is the characteristic function of a set of states paired with their f-value. The state is encoded as usual by a Boolean vector $\vec{v}$ and the f-value is encoded in binary by the Boolean vector $\vec{f}$. Similarly to FSETA*, BDDA* expands all states $min(\vec{v})$ with minimum f-value (f_{min}) in each iteration. The f-value of the child states is computed by arithmetic operations at the BDD level (line 5 and 6). The change in h-value is found by applying a symbolic encoding of the heuristic function to the child and parent state. BDDA* is able to find optimal solutions, but the algorithm only returns the path cost of such solutions. In our implementation, we therefore added a function for tracing a solution backward. In the domains, we have investigated, this extraction function has low complexity, as do those for

GHSETA* and FSETA*. Our investigation of the BDDA* algorithm shows that it often can be improved by

1. defining a computation of $open''$ using a disjunctive partitioned transition relation instead of monolithic transition relation as in line 5,

2. precomputing the arithmetic operation at the end of line 6 for each possible f-value,

3. interleaving the BDD variables of $\vec{f}$, $\vec{e}$, and $\vec{e}'$ to improve the arithmetic BDD operations, and

4. moving this block of variables to the middle of the BDD variable ordering to reduce the average distance to dependent state variables.

The last improvement is actually antagonistic to the recommendation of the BDDA* inventors who locate the $\vec{f}$ variables at the beginning of the variable ordering to simplify the GOLEFT operation. However, we get up to a factor of two speed up with the four modifications above. The algorithms are summarized in the table below.

GHSETA*	:	The GHSETA* algorithm with evaluation function $f(n) = g(n) + h(n)$.
FSETA*	:	The FSETA* algorithm with evaluation function $f(n) = g(n)+h(n)$. This algorithm has been implemented to mimic the BDDA* algorithm. It expands exactly the same states in each iteration. Any performance difference between the two algorithms is due to efficiency differences between state-set branching and the approach used by BDDA*.[2]
BIDIR	:	The BDD-based blind breadth-first bidirectional search algorithm shown in Figure 3.4.
A*	:	Single-state A* with cycle detection, explicit state manipulation, and evaluation function $f(n) = g(n) + h(n)$.
BDDA*	:	The BDDA* algorithm [53] shown in Figure 4.7.
iBDDA*	:	An improved version of BDDA* described below.

In order to factor out differences due to state encodings and BDD computations, all BDD-based algorithms use the same bit vector representation of states, the same variable ordering of the state variables, and similar space allocation and cache sizes of the BDD package. We believe we did an extensive empirical validation. It is necessary since a dissimilarity in just one of the above mentioned properties may cause an exponential performance difference. All algorithms share as many subcomputations as possible, but redundant or unnecessary computations are never carried out for a particular instantiation of an algorithm. The following table shows the measured performance parameters of BIFROST.

t_{total}	:	The total elapsed CPU time of BIFROST.
t_{rel}	:	Time to generate the transition relation. For BDDA* and iBDDA*, this also includes building the symbolic representation of the heuristic function and f-formulas.
t_{search}	:	Time to search for and extract a solution.
$\|sol\|$	:	Solution length.
$\|expand\|$	:	For BIDIR this is the average size of the BDDs representing the search frontier. For FSETA* and GHSETA*, it is the average size of BDDs of search nodes being expanded. For BDDA* and iBDDA*, it is the average size of $open''$.
$\|maxQ\|$	:	Maximum number of queue nodes on the frontier queue.
$\|T\|$	:	The sum of the BDDs representing the partitioned transition relation.
it	:	Number of iterations of the algorithm.

Time is measured in seconds. The time $t_{total} - t_{rel} - t_{search}$ is spent on allocating memory for the BDD package, parsing the problem description and, in case of PDDL problems, analysing the problem in order to make a compact Boolean state encoding. Time out and out of memory are indicated by *Time* and *Mem*. Time out changes between the experiments. The algorithms are out of memory when they start page faulting to the hard drive at approximately 450 MB RAM.

Our experiments cover a wide range of search domains and heuristics. The first domain FG^k is artificial and uses the minimum Hamming distance as heuristic function. It demonstrates that state-set branching may have exponentially better performance than single-state A*. Next, we consider the $D^xV^yM^z$ Puzzle and the 24 and 35 Puzzle using minimum

Hamming distance and sum of Manhattan distance as heuristic function, respectively. We then consider a number of STRIPS planning problems from the AIPS planning competitions [113, 4, 115] using the HSPr heuristic [20] and finally, we study the channel routing problem from VLSI design using a specialized heuristic function.

4.3.1 Search Problems

FG^k

This problem is a modification of Barret and Weld's D^1S^1 problem [8] and has been constructed to show that state-set branching may have exponentially better performance than single-state A*. The problem is easiest to describe in STRIPS. Thus, a state is a set of facts and actions are fact triples defining sets of transitions. The actions are

$$\begin{array}{lll} \begin{array}{lcl} \mathbf{A}_1^1 & & \\ pre & : & \{F^*\} \\ add & : & \{G_1\} \\ del & : & \{\} \end{array} & \begin{array}{lcl} \mathbf{A}_i^1, & i = 2, \cdots, n & \\ pre & : & \{F^*, G_{i-1}\} \\ add & : & \{G_i\} \\ del & : & \{\} \end{array} & \begin{array}{lcl} \mathbf{A}_i^2, & i = 1, \cdots, n & \\ pre & : & \{\} \\ add & : & \{F_i\} \\ del & : & \{F^*\}. \end{array} \end{array}$$

Each action is assumed to have unit cost. The initial state is $\{F^*\}$ and the goal state is $\{G_i | k < i \leq n\}$. Only $\mathbf{A}_i^1$ actions should be applied to reach the goal. Applying an $\mathbf{A}_i^2$ action in any state leads to a wild path since F^* is deleted. The states on wild paths contain F_i facts. Since any subset of F_i facts is possible, the number of states on wild paths grows exponentially with n. The heuristic function is the minimum Hamming distance to the goal states. The only solution is $\mathbf{A}_1^1, \cdots, \mathbf{A}_n^1$ and is non-trivial to find, since the heuristic gives no information to guide the search on the first k steps. Intuitively, the problem can be thought of as walking blindfolded on a sharp ridge for k steps and then with full vision for the remaining $n - k$ steps. A single wrong step has an exponential search penalty of exploring wild paths.

In this experiment, we compare only the total CPU time and number of iterations of GHSETA* and single-state A*. The FG^k problems are defined in NADL$^+$. A specialized poly-time BDD operation for splitting NADL$^+$ actions into transitions with the same search information change is used for GHSETA*. No upper bound ($u = \infty$) is used by GHSETA* and no upper limit of the branching partitions is applied. For the FG^k problems considered, n equals 16. This corresponds to a domain with 2^{33} states. Time out is 600 seconds. The results are shown in Figure 4.8. The performance of A* degrades quickly with the number of unguided steps. A* gets lost expanding an exponentially growing set of states on wild paths. The GHSETA* algorithm is hardly affected by the lack of guidance. The reason is that GHSETA* degenerates to a regular BDD-based blind forward search on the

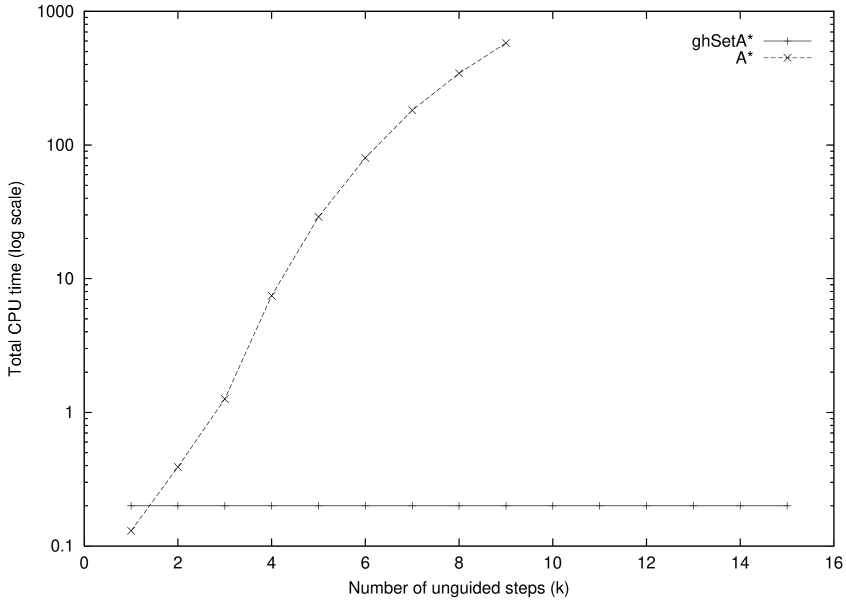


Figure 4.8: Total CPU time of the FG^k problems.

unguided part where the frontier states can be represented by a near symmetric function with polynomial BDD size. Thus, the performance difference between A* and GHSETA* grows exponentially with k.

$D^x V^y M^z$

This problem has the minimum Hamming distance as an admissible heuristic. The domain consists of a set of sliders that can be moved between the corner positions of hypercubes. In any state, a corner position can be occupied by at most one slider. The dimension of the hypercubes is y. There are z sliders of which x are moving on the same cube. The remaining $z-x$ sliders are moving on individual cubes. The sliders are numbered. Initially, they are given corner positions that, when encoded in binary, correspond to an ascending order of their numbers. The goal is to change their positions to a descending order. Each action is assumed to have unit cost. Figure 4.9 shows the initial state of $D^5V^3M^7$.

When $x = z$ all sliders are moving on the same cube. If further $x = 2^y - 1$ all corners of the cube except one will be occupied making it a permutation problem similar to the 8-Puzzle. The key point about this problem is that the x parameter allows the dependency of sliders to be adjusted linearly without changing the size of the domain. For the BDD-based algorithms, the $D^xV^4M^{15}$ problems are defined in NADL$^+$. Again, a specialized poly-time BDD operation for splitting NADL$^+$ actions into transitions with the same search

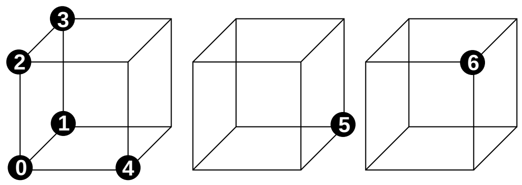


Figure 4.9: The initial state of $D^5V^3M^7$.

information change is applied by GHSETA* and FSETA*. For all problems, the number of states is 2^{60}. For GHSETA* the upper bound for node merging is 200 ($u = 200$). All BDD-based algorithms except BDDA* utilize a disjunctive partitioning with an upper bound on the BDDs representing a partition of 5000. Time out is 500 seconds. For all problems, the BDD-based algorithms use 2.3 seconds on initializing the BDD package ($n = 8M$ and $c = 700K$). The results are shown in Table 4.1. Figure 4.10 shows a graph of the total CPU time for the algorithms.

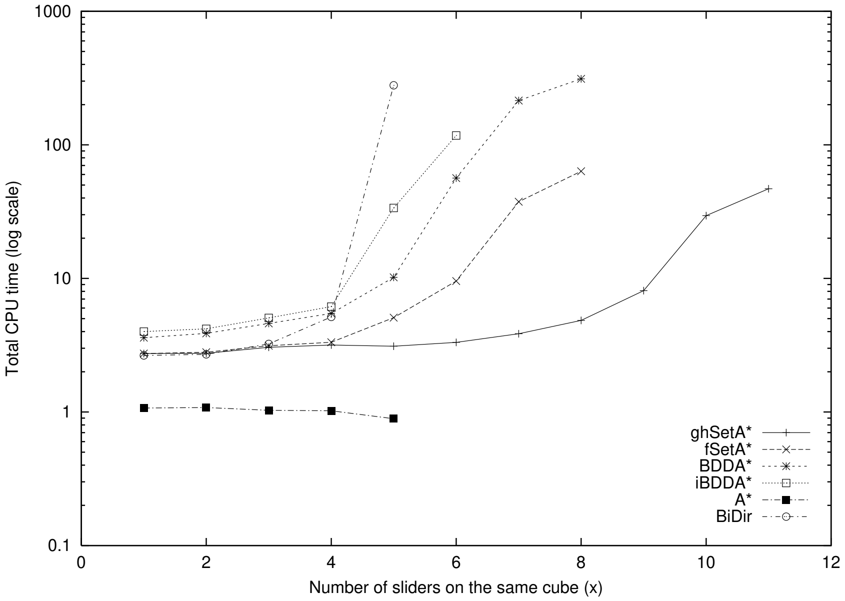


Figure 4.10: Total CPU time of the $D^xV^4M^{15}$ problems.

All solutions found are 34 steps long. Even when the largest number of sliders are on the same cube, a plan with the minimum 34 steps is possible. For BDDA* and iBDDA* the size of the BDD representing the heuristic function is 2014 and 1235, respectively. Both the size of the monolithic and partitioned transition relation grows fast with the dependency of sliders. The problem is that there is no efficient way to model whether a position is occupied or not. The most efficient algorithm is GHSETA*. The FSETA* algorithm has worse performance than GHSETA* because it has to expand all states with minimum f-value in each iteration, whereas GHSETA* focus on a subset of them by having $u = 200$. A subexperiment shows that GHSETA* has similar performance as FSETA* when setting $u = \infty$. The impact of the u parameter is significant for this problem since, even for fairly large values of x, it has an abundance of optimal solutions. BDDA* has much worse performance than FSETA* even though it expands the exact same set of states in each iteration. As we show in Section 4.4, the problem is that the complexity of the computation of $open''$ grows fast with the size of the BDD representing the states to expand. Surprisingly, the performance of iBDDA* is worse than BDDA*. This is unusual, as the remaining experiments will show. The reason might be that only a little space is saved by partitioning the transition relation in this domain. This may cause the computation of $open''$ for iBDDA* to deteriorate because it must iterate through all the partitions. A* performs well when $f(n)$ is a perfect or near perfect discriminator, but it soon gets lost in keeping track of the fast growing number of states on optimal paths. It times out in a *single step* going from about one second to more than 500 seconds. The problem for BIDIR is the usual for blind BDD-based search algorithms applied to hard combinatorial problems: the BDDs representing the search frontiers blow up which increases the time of the image and preimage computations dramatically.

The 24 and 35-Puzzle

We now turn to investigating the $(n^2 - 1)$-Puzzles. The domain consists of an $n \times n$ board with $n^2 - 1$ numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The goal is to reach a configuration where the tiles are ordered ascendingly as shown for the 24-Puzzle in Figure 4.11. For our experiments, the initial state is gener-

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	

Figure 4.11: Goal state of the 24-Puzzle.

Algorithm	x	t_{total}	t_{rel}	t_{search}	$\|expand\|$	$\|Q\|_{max}$	$\|T\|$	it
GHSETA*	1	2.7	0.3	0.2	307.3	33	710	34
	2	2.8	0.3	0.2	307.3	33	1472	34
	3	3.1	0.4	0.3	671.0	33	4070	34
	4	3.2	0.5	0.4	441.7	72	10292	34
	5	3.1	0.4	0.4	194.8	120	20974	34
	6	3.3	0.6	0.4	139.9	212	45978	34
	7	3.9	1.0	0.5	128.4	322	104358	34
	8	4.9	1.9	0.6	115.9	438	232278	34
	9	8.1	5.0	0.8	132.0	557	705956	34
	10	29.5	14.3	12.8	146.1	5103	1970406	373
	11	46.9	43.8	0.8	107.3	336	5537402	34
	12	*Mem*						
FSETA*	1	2.7	0.3	0.2	307.3	1	710	34
	2	2.8	0.3	0.2	307.3	1	1472	34
	3	3.1	0.4	0.4	671.0	1	4070	34
	4	3.3	0.4	0.6	671.0	1	10292	34
	5	5.1	0.5	2.3	1778.6	1	20974	34
	6	9.6	0.6	6.6	2976.5	1	45978	34
	7	37.5	1.0	34.2	9046.7	1	104358	34
	8	63.4	2.0	59.1	9046.7	1	232278	34
	9	408.3	4.9	401.1	24175.4	1	705956	34
	10	*Time*						
BDDA*	1	3.6	0.5	0.4	314.3		355	34
	2	3.9	0.5	0.6	314.3		772	34
	3	4.6	0.6	1.3	678.0		2128	34
	4	5.5	0.8	2.0	678.0		6484	34
	5	10.2	1.3	6.2	1785.6		20050	34
	6	56.4	3.4	50.4	2983.5		64959	34
	7	214.8	10.8	201.1	9053.7		234757	34
	8	312.1	52.7	256.1	9053.7		998346	34
	9	*Time*						
iBDDA*	1	4.0	0.4	0.8	307.3		355	34
	2	4.2	0.4	1.1	307.3		772	34
	3	5.1	0.5	1.9	671.0		2128	34
	4	6.2	0.4	3.0	671.0		6791	34
	5	33.7	0.4	30.4	1778.6		25298	34
	6	117.6	0.5	113.9	2976.5		84559	34
	7	*Time*						
A*	1	1.1				1884		34
	2	1.1				1882		34
	3	1.0				1770		34
	4	1.0				1750		34
	5	0.9				1626		34
	6	*Time*						
BIDIR	1	2.7	0.2	0.1	568.5		355	34
	2	2.7	0.2	0.2	630.8		772	34
	3	3.2	0.3	0.7	2305.1		2128	34
	4	5.2	0.2	2.6	3131.1		5159	34
	5	278.9	0.2	276.4	30445.0		10610	34
	6	*Time*						

Table 4.1: Results of the $D^x V^4 M^{15}$ problems.

ated by performing r random moves from the goal state.[3] We assume unit cost transitions and use the usual sum of Manhattan distances of the tiles to their goal position as heuristic function. This heuristic function is admissible. For GHSETA* and FSETA* a disjunctive branching partitioning is easy to compute since δh of an action changing the position of a single tile is independent of the position of the other tiles. The two algorithms have no upper bound on the size of BDDs in the frontier nodes ($u = \infty$). For the BDD-based algorithms, the problems are defined in NADL$^+$ and the best results are obtained when having no limit on the partition size. Thus, BDDA*, iBDDA*, and BIDIR use a monolithic transition relation. The number of states for the 24-Puzzle is 2^{125}. The results of this problem are shown in Table 4.2. For all 24-Puzzle problems, the BDD-based algorithms spend 3.6 seconds on initializing the BDD package ($n = 15M$ and $c = 500K$). Time out is 10000 seconds. For BDDA* and iBDDA* the size of the BDD representing the heuristic function is 33522 and 18424, respectively. For GHSETA* and FSETA* the size of the transition relations is 70582, while the size of the transition relation for BDDA* and iBDDA* is 66673. Thus, a small amount of space was saved by using a monolithic transition relation representation. However, GHSETA* and FSETA* have better performance than BDDA* and iBDDA* mostly due to the their more efficient node expansion computation. Interestingly, both BDDA* and iBDDA* spend significant time computing the heuristic function in this domain. The GHSETA* and FSETA* also scale better than A* and BIDIR. A* has good performance because it does not have the substantial overhead of computing the transition relation and finding actions to apply. However, due to the explicit representation of states, it runs out of memory for solution depths above 50. For BIDIR, the problem is the usual: the BDDs representing the search frontiers blow up. Figure 4.12 shows a graph of the total CPU time of the 24 and 35-puzzle. Again time out is 10000 seconds.

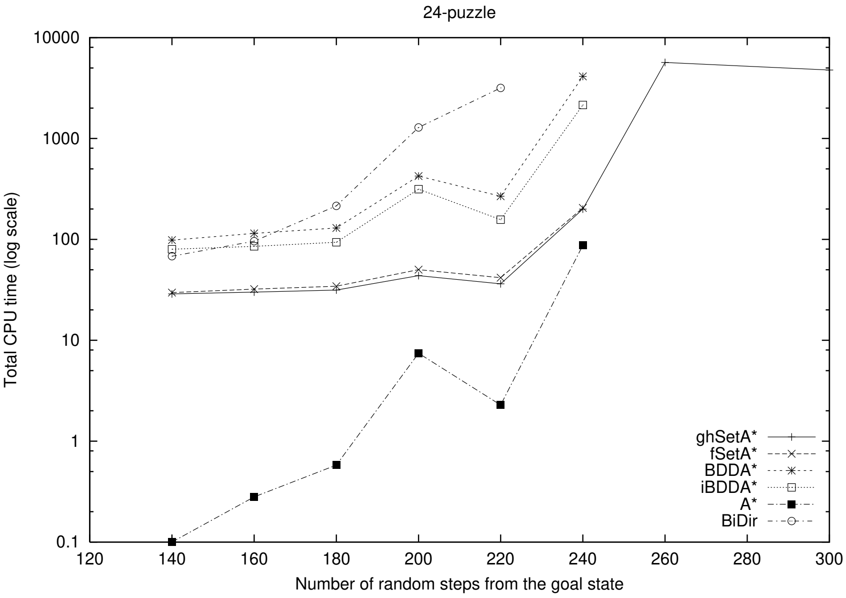
4.3.2 Planning Problems

In this section, we consider four planning problems from the STRIPS track of the AIPS 1998 [113], 2000 [4], and 2002 [115] planning competition. The problems are defined in the STRIPS fraction of PDDL. The reachability analysis necessary to compactly encode STRIPS domains described in Section 3.1 is based an approach described in [48]. It is fast for the problems considered in experimental evaluation (for most problems less than 0.04 seconds). The algorithm proceeds in a breadth-first manner such that each ground predicate or *fact* f can be assigned a depth $d(f)$ where it is reached. Similar to the MIPS planning system [48], we use this measure to approximate the HSPr heuristic [20]. HSPr is an efficient but non-admissible heuristic for backward search. For a state given by a set of

[3] In each of these steps choosing the move back to the previous state is illegal.

Algorithm	r	t_{total}	t_{rel}	t_{search}	$\|sol\|$	$\|expand\|$	$\|Q\|_{max}$	it
GHSETA*	140	28.8	22.1	2.7	26	187.5	23	93
	160	30.0	22.2	3.8	28	213.2	24	175
	180	31.4	22.2	5.3	32	270.2	28	253
	200	43.7	21.9	14.9	36	786.2	31	575
	220	36.3	22.2	10.1	36	411.1	31	490
	240	199.3	22.0	173.2	50	2055.5	44	1543
	260	5673.7	23.9	5644.5	56	10641.2	48	2576
	280	*Mem*						
	300	4772.7	20.9	4743.97	60	9761.3	53	2705
	320	*Mem*						
FSETA*	140	29.7	21.0	4.7	26	669.9	1	42
	160	32.2	20.9	7.4	28	1051.6	1	57
	180	34.3	21.0	9.5	32	1207.0	1	69
	200	50.1	21.0	25.3	36	5276.0	1	93
	220	41.8	21.0	17.0	36	3117.6	1	88
	240	205.2	21.0	180.5	50	18243.3	1	156
	260	*Mem*						
BDDA*	140	98.5	83.0	11.3	26	676.9		42
	160	114.7	83.2	27.4	28	1058.6		57
	180	129.8	82.9	42.7	32	1214.0		69
	200	425.0	83.1	337.1	36	5283.0		93
	220	267.7	82.8	180.6	36	3124.6		88
	240	4120.1	83.1	4032.8	50	18250.3		156
	260	*Time*						
iBDDA*	140	79.8	66.7	5.9	26	669.9		42
	160	85.3	65.7	11.8	28	1051.6		57
	180	93.6	65.7	20.0	32	1207.0		69
	200	314.6	65.8	240.9	36	5276.0		93
	220	156.9	65.6	83.5	36	3117.6		88
	240	2150.3	65.9	2076.6	50	18243.3		156
	260	*Mem*						
A*	140	0.1			26		300	221
	160	0.9			28		725	546
	180	0.6			32		1470	1106
	200	7.4			36		15927	12539
	220	2.3			36		5228	4147
	240	87.1			50		159231	133418
	260	*Mem*						
BIDIR	140	68.1	36.6	27.9	26	34365.2		26
	160	96.0	36.8	55.6	28	55388.4		28
	180	214.7	36.8	174.3	32	106166.0		32
	200	1286.0	36.8	1245.6	36	359488.0		36
	220	3168.8	36.8	3128.4	36	421307.0		36
	240	*Mem*						

Table 4.2: Results of the 24-Puzzle problems.



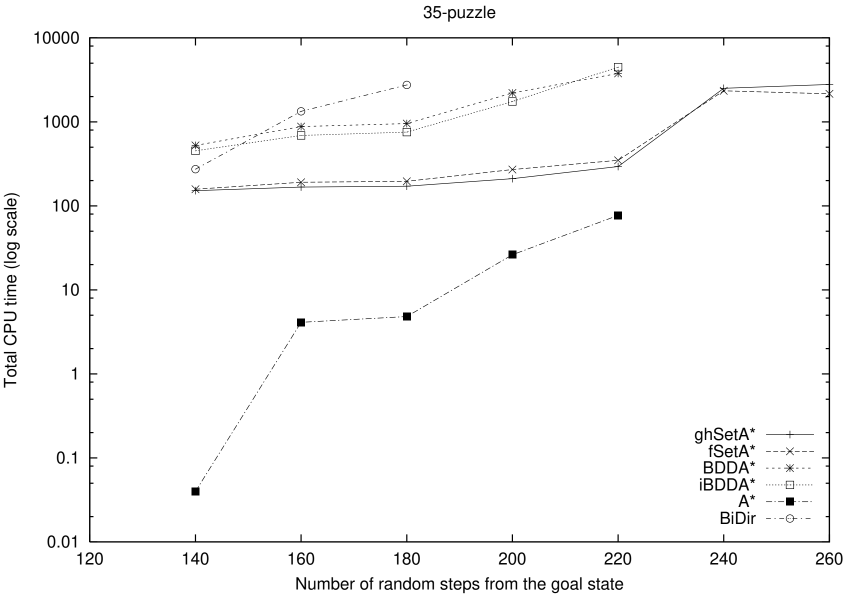


Figure 4.12: Total CPU time for the 24 and 35-Puzzle problems.

facts S, the approximation to HSPr is given by

$$h(S) = \sum_{f \in S} d(f).$$

[4] A disjunctive branching partitioning for this heuristic is efficient to generate given that each action (pre, add, del) leading from S to $S' = (S \cup add) \setminus del$ satisfies

$$del \subseteq pre \quad \text{and} \quad add \cap pre = \emptyset.$$

These requirements are natural and satisfied by all the planning domains considered in this experimental evaluation. Due to the constraints, we get

$$\begin{aligned}\delta h &= h(S') - h(S) \\ &= h(add \setminus S) - h(del) \\ &= \sum_{f \in add \setminus S} d(f) - \sum_{f \in del} d(f).\end{aligned}$$

Thus, each action is partitioned in up to $2^{|add|}$ sets of transitions with different δh-value. In order to simplify the computation of the initial heuristic value, all problems have been modified to a single goal state. Furthermore, in domains where the HSPr approximation either systematically under or over estimates the true remaining cost, we have scaled it accordingly.

Blocks World

The Blocks World is a classical planning domain. It consists of a set of cubic blocks sitting on a table. A robot arm can stack and unstack blocks from some initial configuration to a goal configuration. The problems, we consider, are from the STRIPS track of the AIPS 2000 planning competition. The number of states grows from 2^{17} to 2^{80}. The HSPr heuristic is scaled by a factor of 0.4. The GHSETA* and FSETA* algorithms have no upper bound on the size of BDDs of the nodes on the frontier ($u = \infty$). For all BDD-based algorithms, the partition limit was 5000. For each problem, these algorithms spend about 2.5 seconds on initializing the BDD package ($n = 8M$ and $c = 800K$). Time out is 500 seconds in all experiments. The results are shown in Table 4.3. The top graph of Figure 4.13 shows the total CPU time of the algorithms.

For BDDA* and iBDDA* the size of the BDD representing the heuristic function is in the range of $[8, 1908]$ and $[8, 1000]$, respectively. The GHSETA* and FSETA* algorithms have significantly better performance than all other algorithms. As usual BDDA* and iBDDA* suffer from an inefficient expansion computation while the frontier BDDs blow

[4]This is an approximation to the HSPr heuristic since the HSPr heuristic for a fact f estimates the number of *actions* needed to produce f from the initial state if the the delete set of actions is ignored. By measuring the depth of f in a forward reachability analysis from the initial state, we only consider the depth of this dependency tree of actions.

Algorithm	p	t_{total}	t_{rel}	t_{search}	$\|sol\|$	$\|expand\|$	$\|Q\|_{max}$	it	$\|T\|$
GHSETA*	4	2.6	0.0	0.0	6	19.5	1	6	706
	5	2.7	0.1	0.1	12	33.4	11	31	1346
	6	2.6	0.1	0.1	12	57.7	9	30	2608
	7	3.1	0.2	0.4	20	53.8	48	152	4685
	8	4.1	0.3	1.3	18	540.4	12	72	7475
	9	17.0	0.4	14.1	32	331.8	94	991	8717
	10	116.2	0.6	113.1	38	744.9	111	2309	11392
	11	133.5	0.7	130.2	32	1404.9	91	1200	16122
	12	14.8	1.0	11.2	34	410.3	120	557	18734
	13	*Time*							
	14	112.1	1.7	107.8	38	1067.8	125	1061	30707
	15	*Time*							
FSETA*	4	2.5	0.0	0.0	6	29.8	1	6	706
	5	2.7	0.1	0.1	12	68.7	4	23	1346
	6	2.7	0.1	0.1	12	126.8	2	20	2608
	7	3.2	0.2	0.5	20	121.9	8	92	4685
	8	3.9	0.3	1.1	18	1328.8	2	35	7475
	9	30.0	0.4	27.1	32	935.5	10	610	8717
	10	217.0	0.6	213.8	38	2594.4	12	1098	11392
	11	259.8	0.8	256.4	32	4756.0	9	671	16122
	12	39.2	1.0	35.7	34	817.0	13	860	18734
	13	*Time*							
	14	274.3	1.7	270.0	38	1555.1	13	1462	30707
	13	*Time*							
BDDA*	4	3.3	0.0	0.1	6	37.8		6	706
	5	3.6	0.2	0.2	12	76.7		23	1365
	6	3.6	0.2	0.2	12	134.8		20	2334
	7	4.9	0.5	1.2	20	129.9		92	4669
	8	6.0	0.5	2.2	18	1336.8		35	6959
	9	100.8	1.1	96.5	32	943.5		610	9923
	10	*Time*							
iBDDA*	4	2.7	0.0	0.0	6	29.8		6	706
	5	2.8	0.1	0.1	12	68.7		23	1365
	6	2.9	0.1	0.1	12	126.8		20	2334
	7	3.7	0.3	0.7	20	121.9		92	4669
	8	6.2	0.4	3.2	18	1328.8		35	7123
	9	113.7	0.6	110.3	32	935.5		610	10361
	10	*Time*							
A*	4	0.0		0.0	6		8	15	
	5	0.2		0.2	12		62	70	
	6	0.4		0.4	12		115	102	
	7	1.3		1.2	20		287	287	
	8	31.9		31.6	18		7787	5252	
	9	233.9		232.9	32		38221	31831	
	10	*Time*							
BIDIR	4	2.6	0.0	0.0	6	124.5		6	706
	5	2.6	0.1	0.0	12	228.3		12	1423
	6	2.7	0.1	0.1	12	438.8		12	2567
	7	3.6	0.2	0.8	20	1931.3		20	5263
	8	9.7	0.3	6.8	18	11181.8		18	8157
	9	146.8	0.4	143.9	30	75040.9		30	11443
	10	*Time*							

Table 4.3: Results of the Blocks World problems.

up for BIDIR. The general A* algorithm for STRIPS planning problems is less domain-tuned than the previous A* implementations. In particular, it must check the precondition of all actions in each iteration in order to find the ones that are applicable. This may explain the poor performance of A*.

Gripper

The Gripper problems are from the first round of the STRIPS track of the AIPS 1998 planning competition. The domain consists of two rooms, A and B, connected with a door and robot with two grippers. Initially, a number of balls are located in room A, and the goal is to move them to room B. The number of states grows linearly from 2^{12} to 2^{88}. The GHSETA* and FSETA* algorithms have no upper bound on the size of BDDs in the frontier nodes ($u = \infty$). For all BDD-based algorithms no partition limit is used, and they spend about 0.8 seconds on initializing the BDD package ($n = 2M$ and $c = 400K$). All algorithms generate optimal solutions. The results are shown in Table 4.4. The bottom graph of Figure 4.13 shows the total CPU time of the algorithms. Interestingly, BIDIR is the fastest algorithm in this domain since the BDDs representing the search frontier only grow moderately during the search. The GHSETA* and FSETA* algorithms, however, have almost as good performance. BDDA* and iBDDA* has particularly bad performance in this domain. The problem is that the BDDs of their frontier nodes are large compared to other domains and that the expansion computation of these algorithms seems to scale poorly. We will investigate this problem in detail in Section 4.4.

Logistics

The logistics domain considers moving packages with trucks between sub-cities and with airplanes between cities. The problems considered are from the STRIPS track of the AIPS 2000 planning competition. The number of states grows from 2^{21} to 2^{86}. The GHSETA* and FSETA* algorithms have no upper bound on the size of BDDs in the frontier nodes ($u = \infty$). For all BDD-based algorithms, a partition limit of 5000 is used and they spend about 2.0 seconds on initializing the BDD package ($n = 8M$ and $c = 400K$). Due to systematic under estimation, the HSPr heuristic is scaled with a factor of 1.5. The top graph of Figure 4.14 shows the total CPU time of the algorithms.

ZenoTravel

ZenoTravel is from the STRIPS track of the AIPS 2002 planning competition. It involves transporting people around in planes, using different modes of movement: fast and slow.

Algorithm	p	t_{total}	t_{rel}	t_{search}	$\|expand\|$	$\|Q\|_{max}$	it	$\|T\|$
GHSETA*	2	0.9	0.1	0.02	68.8	5	21	594
	4	1.0	0.1	0.08	168.9	6	43	1002
	6	1.3	0.2	0.27	314.9	6	65	1410
	8	1.5	0.3	0.34	504.8	6	87	1818
	10	1.8	0.4	0.54	738.1	6	109	2226
	12	2.3	0.5	0.88	1014.7	6	131	2634
	14	3.0	0.7	1.33	1334.5	6	153	3042
	16	3.6	0.9	1.78	1697.5	6	175	3450
	18	4.5	1.1	2.46	2103.7	6	197	3858
	20	5.7	1.4	3.37	2553.1	6	219	4266
FSETA*	2	1.0	0.1	0.1	95.4	1	17	594
	4	1.0	0.1	0.1	231.2	1	29	1002
	6	1.2	0.2	0.2	423.9	1	41	1410
	8	1.6	0.3	0.3	673.4	1	53	1818
	10	2.0	0.4	0.6	979.9	1	65	2226
	12	2.5	0.6	1.0	1343.3	1	77	2634
	14	3.1	0.8	1.4	1763.5	1	89	3042
	16	3.7	0.9	1.9	2240.7	1	101	3450
	18	5.0	1.2	2.9	2774.7	1	113	3858
	20	5.7	1.5	3.2	3365.6	1	125	4266
BDDA*	2	1.8	0.1	0.2	103.4		17	323
	4	2.4	0.2	0.6	239.2		29	539
	6	3.4	0.3	1.5	431.9		41	755
	8	6.1	0.6	4.0	681.4		53	971
	10	16.9	0.9	14.4	987.9		65	1187
	12	40.7	1.2	37.9	1351.3		77	1403
	14	81.7	1.6	78.5	1771.5		89	1619
	16	149.3	2.2	145.4	2248.7		101	1835
	18	240.4	3.1	235.5	2782.7		113	2051
	20	391.1	3.9	385.5	3373.6		125	2267
iBDDA*	2	1.2	0.1	0.1	95.4		17	323
	4	1.6	0.1	0.4	231.2		29	539
	6	2.3	0.3	1.0	423.9		41	755
	8	3.6	0.4	2.2	673.4		53	971
	10	6.2	0.6	4.5	979.9		65	1187
	12	12.2	0.9	9.2	1343.3		77	1403
	14	23.5	1.1	21.3	1763.5		89	1619
	16	44.8	1.6	42.1	2240.7		101	1835
	18	76.1	2.2	72.4	2774.7		113	2051
	20	120.9	2.7	116.7	3365.6		125	2267
A*	2	3.9		3.9		698	1286	
	4	422.9		422.3		26434	85468	
	6	*Time*						
BIDIR	2	0.9	0.1	0.0	125.4		17	323
	4	1.0	0.1	0.1	290.9		29	539
	6	1.2	0.2	0.1	589.7		41	755
	8	1.4	0.3	0.3	958.2		53	971
	10	1.7	0.4	0.5	1404.3		65	1187
	12	2.2	0.5	0.8	1611.0		77	1403
	14	2.6	0.7	1.0	2025.6		89	1619
	16	3.2	0.9	1.3	3265.6		101	1835
	18	3.8	1.2	1.7	4074.4		113	2051
	20	4.5	1.5	2.1	4944.9		125	2267

Table 4.4: Results of the Gripper problems.

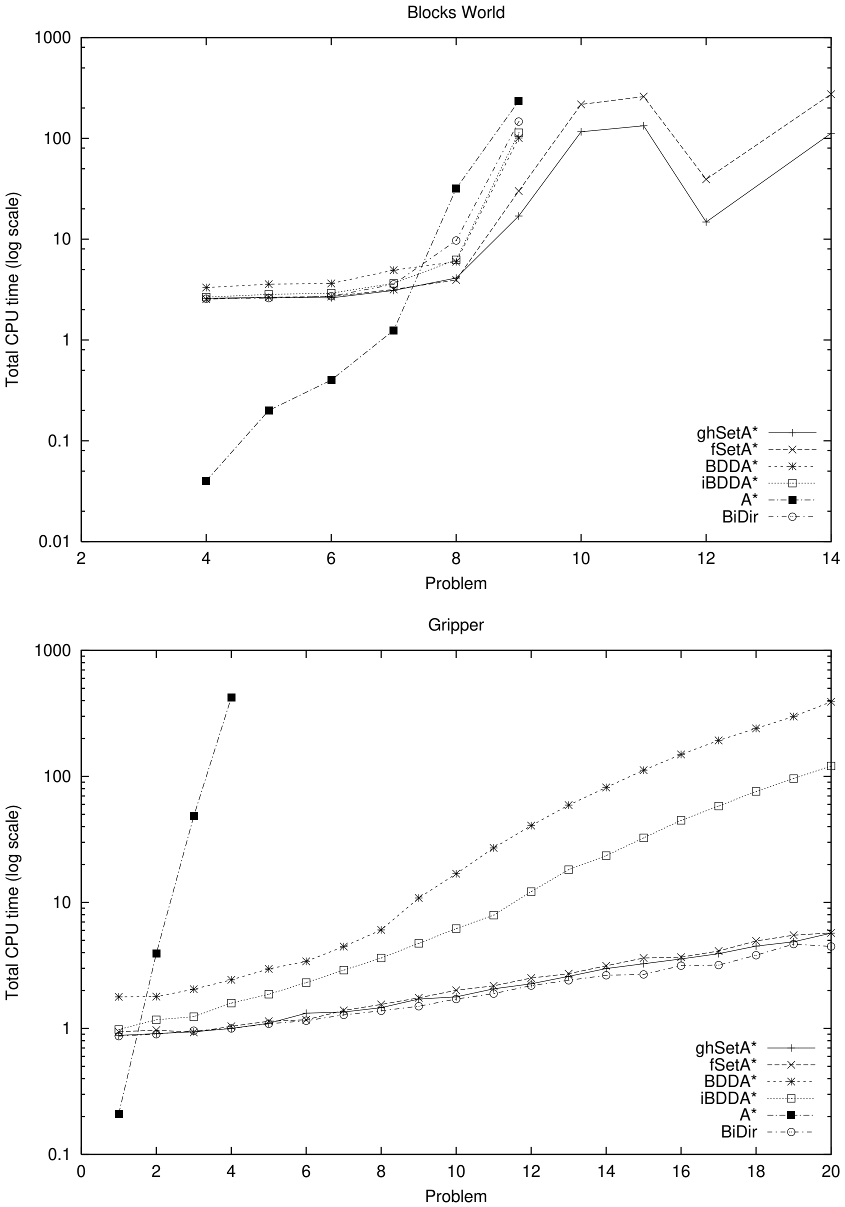


Figure 4.13: Total CPU time for the Blocks World and Gripper problems.

The number of states grows from 2^9 to 2^{165}. The GHSETA* and FSETA* algorithms have no upper bound on the size of BDDs in the frontier nodes ($u = \infty$). For all BDD-based algorithms a partition limit of 4000 is used. About 2.7 seconds is spent on initializing the BDD package ($n = 10M$ and $c = 700K$). The bottom graph of Figure 4.14 shows the total CPU time of the algorithms. The results are very similar to the results of the logistics problems.

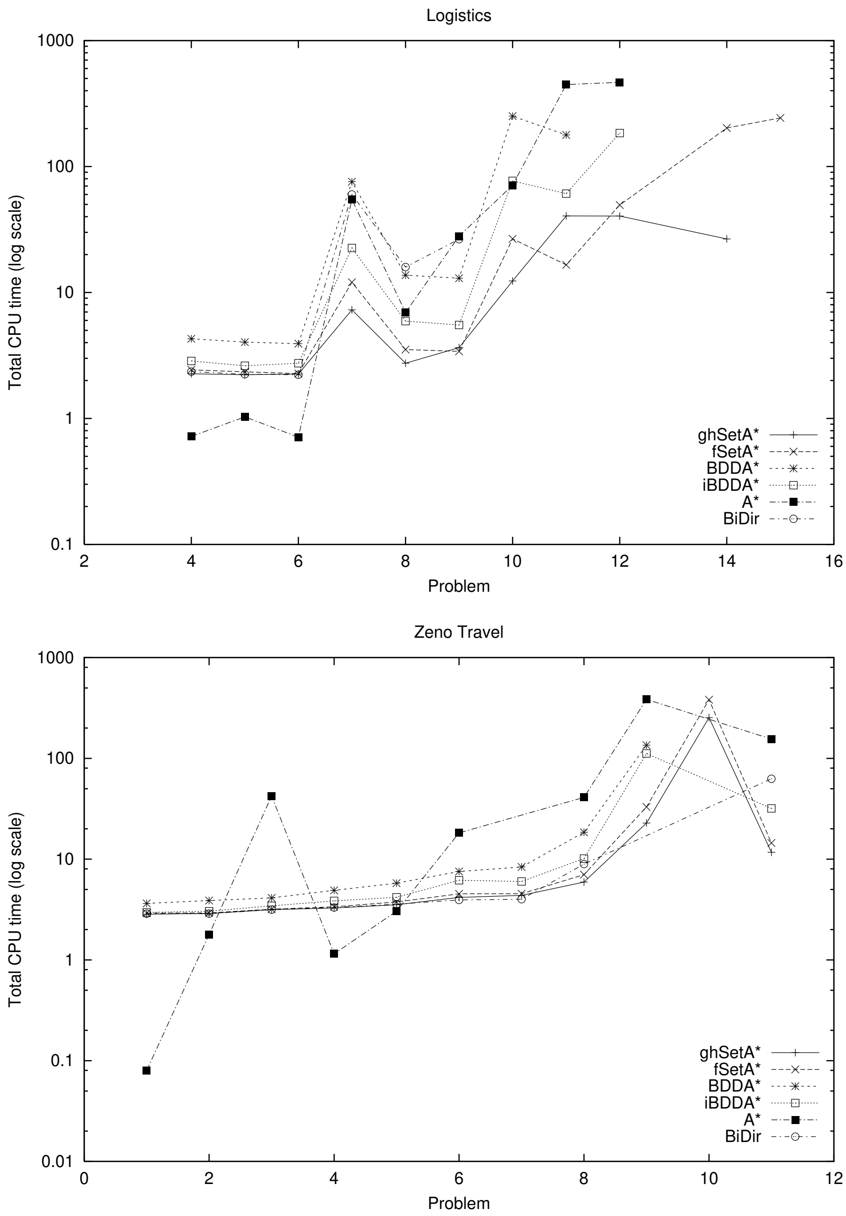


Figure 4.14: Total CPU time for the Logistics and ZenoTravel problems. Problem 10 of ZenoTravel can only be solved by GHSETA* and FSETA*.

4.3.3 Channel Routing Problems

Channel routing is a fundamental subtask in the layout process of VLSI-design. It is an NP-complete problem which makes exact solutions hard to produce. Channel routing considers connecting pins in the small gaps or channels between the cells of a chip. In its usual formulation, two layers are used for the wires: one where wires go horizontal (tracks) and one where wires go vertical (columns). In order to change direction, a connection must be made between the two layers. These connections are called vias. Pins are at the top and bottom of the channel. A set of pins that must be connected is called a net. The problem is to connect the pins optimally according to some cost function. The cost function studied here equals the total number of vias used in the routing. Figure 4.15 shows an example of an optimal solution to a small channel routing problem. The cost of the solution is 4. One way to apply search to solve a channel routing problem is to route the nets from

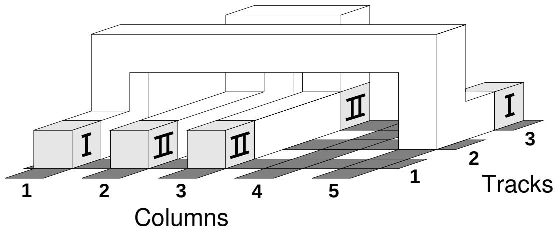


Figure 4.15: A solution to a channel routing problem with 5 columns, 3 tracks, and 2 nets (labeled I and II). The pins are numbered according to what net they belong.

left to right. A state in this search is a column paired with a routing of the nets on the left side of that column. A transition of the search is a routing of live nets over a single column. Recently, it has been shown that BDD-based channel routing algorithms utilizing this strategy efficiently can scale in the number of columns [153, 160]. The belief is that such algorithms can be used to perform subcomputations of a global router that decomposes the routing into a vertical and horizontal part.

A* can be used in the usual way to find optimal solutions. An admissible heuristic function for our cost function is the sum of the cost of routing all remaining nets optimally ignoring interactions with other nets. We have implemented a specialized search engine to solve channel routing problems with GHSETA* [90]. The key point about this application of state-set branching is that GHSETA* utilizes a conjunctive branching partitioning instead of a disjunctive branching partitioning as in all other experiments reported so far. This is possible since a transition can be regarded as the joint result of routing each net in turn.

The performance of GHSETA* is evaluated using problems produced from two ISCAS-85 circuits [160]. For each of these problems the parameters of the BDD package are hand tuned for best performance. There is no upper bound on the size of BDDs in frontier nodes ($u = \infty$) and no limit on the size of the partitions. Time out is 600 seconds. Table 4.5 shows the results. The performance of GHSETA* is similar to previous applications of BDDs to channel routing [153, 160, 175]. However, in contrast to previous approaches, GHSETA* finds optimal solutions, whereas the previous algorithms only find valid solutions. The

Circuit	$c - t - n$	t_{total}	t_{rel}	t_{search}	$\|Q\|_{max}$	it
Add	38-3-10	0.2	0.1	0.2	1	40
	47-5-27	0.8	0.7	0.1	24	46
	41-3-12	0.2	0.1	0.1	1	42
	46-7-20	5.0	3.5	1.5	56	89
	25-4-6	0.1	0.0	0.1	1	30
C432	83-4-33	0.4	0.2	0.2	0	93
	89-11-58	*Mem*				
	101-9-57	286.1	61.5	206.6	135	113
	99-8-58	34.0	13.5	20.5	59	448
	97-10-63	295.0	99.7	195.3	129	109
	101-7-53	15.7	11.5	4.2	90	101
	95-9-48	223.8	58.9	164.9	59	399
	95-10-48	*Time*				
	84-5-23	3.2	0.7	2.5	0	92

Table 4.5: Results of the ISCAS-85 channel routing problems. A problem, $c - t - n$, is identified by its number of columns (c), tracks (t), and nets (n).

experimental results, however, show that the benefit of using guided BDD-based search for channel routing is limited. The reason is that the BDDs representing the search frontier do not blow up in this domain as in most other planning domains. Instead the intermediate BDDs of the image computation blow up, both when this computation is based on a regular conjunctive partitioning for blind search and when it is utilizing a conjunctive branching partitioning for guided search. It may be the case, though, that more efficient encodings of channel routing domains exist. For instance, for the $(n^2 - 1)$-Puzzles, it is much more efficient to encode for each position what tile it holds rather than for each tile encode what its position is. The former encoding is redundant compared to the latter because it also represents the position of the blank space. However, the representation of actions is substantially simplified in the former encoding since the position of the blank space is known.

4.4 Conclusion

We conclude this chapter by comparing state-set branching to single-state heuristic search, blind BDD-based search, and BDDA*.

State-Set Branching versus Single-State Heuristic Search

Heuristic search is trivial if the heuristic function is very informative. In this case, state-set branching may have worse performance than single-state heuristic search due to the overhead of computing the transition relation. Thus, we do not expect state-set branching algorithms to have better performance than the single-state heuristic search algorithms applied in the AIPS planning competitions because the problems considered have very strong heuristics [79]. In this experimental evaluation, we consider finding optimal or near optimal solutions with state-set branching implementations of A*. The studied heuristic functions are classical but leave a significant search element for the algorithms to handle. For these problems, state-set branching outperforms single-state A*. Notice that this result is consistent with the fact that single-state A* is optimally efficient. The reason is that a state-set branching implementation of A* may use an exponentially more compact state representation than single-state A*.

State-Set Branching versus Blind BDD-based Search

Blind BDD-based search has been successfully applied in symbolic model checking and circuit verification. It has been shown that many problems encountered in practice are tractable when using BDDs [168]. The classical search problems studied in AI, however, seem to be harder and have longer solutions than the problems considered in formal verification. When applying blind BDD-based search to these problems, the BDDs used to represent the search frontier often grow fast. The experimental evaluation of state-set branching shows that this problem can be substantially reduced when efficiently splitting the search frontier according to a heuristic evaluation of the states.

State-Set Branching versus BDDA*

State-set branching implementations of A* such as GHSETA* and FSETA* are fundamentally different from BDDA*. BDDA* does not exploit a partitioning of the transitions according to how they change the g and h-value. Instead, it imitates the usual explicit application of the heuristic function via a symbolic computation. It would be reasonable to expect that the symbolic representation of practical heuristic functions often is very large.

However, this is seldom the case for the heuristic functions studied in this experimental evaluation. The major challenge for BDDA* is that the arithmetic computations at the BDD level scales poorly with the size of the BDD representing the set of states to expand (line 5 and 6 in Figure 4.7). This hypothesis can be empirically verified by measuring the CPU time used by FSETA* and iBDDA* to expand a set of states. Recall that FSETA* and iBDDA* expand the exact same set of states in each iteration. Any performance difference is therefore solely caused by their expansion techniques. The results are shown in Figure 4.16. The reported CPU time is the average of the 15-Puzzle with 50, 100, and 200 random steps, Logistics problem 4 to 9, Blocks World problem 4 to 9, Gripper problem 1 to 20, and DxV4M15 with x varying from 1 to 6. For very small frontier BDDs, iBDDA* is

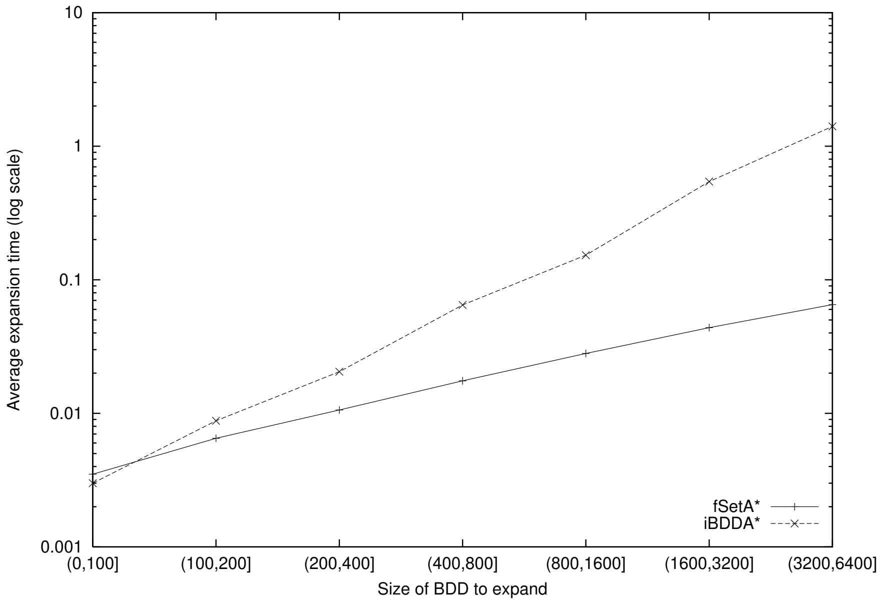


Figure 4.16: Node expansion times of FSETA* and BDDA*.

slightly faster than FSETA*. This is probably because small frontier BDDs mainly are generated by easy problems where a possibly monolithic transition relation used by iBDDA* is more efficient than the partitioned transition relation used by FSETA*. However, for large frontier BDDs, BDDA* needs much more time to expand the frontier than FSETA*. Another limitation of BDDA* is the inflexibility of BDD-based arithmetic. It makes it hard to extend BDDA* efficiently to general evaluation functions and arbitrary transitions costs.

4.5 Summary

This chapter has introduced a new framework called *state-set branching*. State-set branching seamlessly combines BDDs and heuristic search via a state-set version of the classical best-first search algorithm using a new partitioning technique called *branching partitioning*. It has been shown that the framework is general. It applies to any heuristic function, any cost function and any node evaluation function. In addition, both disjunctive and conjunctive versions of branching partitions can be defined. The experimental evaluation proves state-set branching to be a powerful approach that often outperforms both single-state heuristic search and blind BDD-based search. Moreover, it has substantially better performance than the approach used by BDDA*.

Chapter 5

Non-Deterministic State-Set Branching

A limitation of the current BDD-based non-deterministic planning algorithms is that they perform blind search. A backward search frontier is expanded in a breadth-first manner and the final non-deterministic plan may cover a large number states that are unreachable from the initial states. In this chapter, we describe how to use state-set branching to guide these algorithms [95]. We begin in Section 5.1 by introducing a generic non-deterministic planning algorithm for guided search. Then, in Section 5.2, the guided precomponents of weak, strong cyclic, and strong planning are defined. Section 5.3 describes a range of experimental results showing that the new algorithms may dramatically reduce both the search time and the plan size compared with the current algorithms. Finally, Section 5.4 draws conclusions.

5.1 Guided Non-Deterministic Planning

As described in the previous chapter, the state-set branching framework has two independent parts: a modification of the best-first search algorithm to expanding sets of states in each iteration and a specialized partitioning technique called *branching partitioning* to implement the new algorithm efficiently with BDDs. A key observation is that branching partitioning also can be used to propagate search control information between states in non-deterministic domains. This follows directly from the fact that branching partitioning is defined at the transition level and therefore is independent of whether actions are deterministic. The major difference when considering non-deterministic planning is that it seems to be very hard, if not impossible, to cast the generic non-deterministic planning algorithm NDP shown in Figure 3.8 as a search tree algorithm. The problem is to guarantee completeness for strong and strong-cyclic planning. Consider for example using a search tree to generate a strong plan. Assume that the algorithm at some point during the search adds a node n with states P from the frontier of the search tree to the plan. Let C denote the set of states covered by the plan after this incrementation. The algorithm then computes the child nodes of n. This can be done by finding state-action pairs (SAs) that can reach P in one step and form a subset of a strong precomponent of C. Assume that the algorithm in the next iteration expands a node that is not a child node of n. Let C' denote the set of states covered by the plan after adding the states of this node to the plan. This, however, may affect the child nodes of n. If the child nodes are recomputed with respect to C' and not C they may contain a larger set of SAs since $C' \supseteq C$. This makes the algorithm incomplete since the child nodes of n are computed only once. A similar problem exists for strong-cyclic planning. For weak planning, on the other hand, it is possible to define a complete search tree algorithm since the set of SAs of the child nodes are independent of the set of states in the plan (defined in [95]). In this presentation, though, we propose a general framework for pure heuristic non-deterministic planning called *non-deterministic state-set branching* where a heuristic function is used to select a subset of the blind precomponent in each iteration. Non-deterministic state-set branching is based on the generic guided non-deterministic planning algorithm GNDP shown in Figure 5.1.

function GNDP(s_0, G, h_g)
$P \leftarrow \emptyset$; $\mathbf{C} \leftarrow \mathit{emptyMap}$; $\mathbf{C}[h_g] \leftarrow G$
while $s_0 \notin C$
  $\mathbf{P_c} \leftarrow$ GPRECOMP$(\mathbf{C})$
  if $|\mathbf{P_c}| = 0$ **then return** "no solution exists"
  $P \leftarrow P \cup P_c$
  for $k = 1$ **to** $|\mathbf{P_c}|$
    $\mathbf{C}[h_k] \leftarrow \mathbf{C}[h_k] \cup$ STATES$(\mathbf{P_c}[h_k])$
return P

Figure 5.1: A generic guided algorithm for synthesizing non-deterministic plans.

The GNDP algorithm is similar to NDP. The main difference is that it keeps the set of states covered by the plan in a map $\mathbf{C}$. The purpose of the map is to partition the covered states with respect to the value of a heuristic function that for a state s estimates the minimum length of a path from s_0 to s. In each iteration, a guided precomponent $\mathbf{P_c}$ is computed and added to the plan. The precomponent function must be valid according to Definition 5.1.

Definition 5.1 (Guided Precomponent Function) *If* $\mathbf{C}$ *associates states with their correct* h*-value then a guided precomponent function* GPRECOMP$(\mathbf{C})$ *is valid iff*

- GPRECOMP : $(\mathbb{R}_0^+ \rightarrow 2^S) \rightarrow (\mathbb{R}_0^+ \rightarrow 2^{S \times Act})$,
- GPRECOMP$(\mathbf{C})$ *terminates,*
- *if* $\mathbf{P_c} = $ GPRECOMP$(\mathbf{C})$ *then for any* $\langle s, a \rangle \in \mathbf{P_c}[h_i]$ *, we have* $s \notin C$ *and the* h*-value of* s *is* h_i*.*

The completeness of the algorithm is due to the fact that the precomponent computation does not rely on previous computations that may have become outdated. A limitation of GNDP is that all goal states are assumed to have identical h-value (h_g). It is easy, though, to generalize the algorithm to take a set of goal states partitioned with respect to h-value as input. It has not been done here to simplify the presentation.

Theorem 5.1 (Termination of GNDP) GNDP *terminates.*

Proof. Given in Appendix B □

5.2 Guided Precomponents

In this section, we introduce the guided precomponents for weak, strong cyclic, and strong planning used by GNDP. For weak and strong planning, the guided precomponent is the set of state-action pairs (SAs) in a complete blind precomponent that has states with minimum h-value. In both cases, this strategy results in a pure heuristic search, since only the heuristic estimate of the distance to the initial state is used to guide the search. For strong-cyclic planning, the guided precomponent is computed from a set of candidate SAs built from a search tree of weak precomponents grown from the set of states covered by the plan. In order to avoid a too "narrow" candidate set, both the heuristic value of the states and the depth in the tree is taken into account when choosing a node to expand. After each expansion of the candidate set, the SCPLANAUX function defined in Section 3.2.2 is used to extract a strong cyclic precomponent from the candidate, if possible.

Similarly to blind non-deterministic planning, the core operation in guided non-deterministic planning is to find the preimage of a set of states. However, we also need to split the SAs in the preimage according to the h-value of the states. As for deterministic state-set branching, we can use a disjunctive branching partitioning to compute and split the preimage in a single operation. Assume that the disjunctive branching partitioning is of the form

$$R_1(\vec{v}, \vec{a}, \vec{y}'_1), \cdots, R_n(\vec{v}, \vec{a}, \vec{y}'_n)$$

where the h-value change (in the is forward direction) of transitions in subrelation i is δh_i. The preimage of subrelation i is then given by

$$\textsc{PreImgSA}_i(C) \;\equiv\; \exists \vec{y}\,'_i \,.\; R_i(\vec{v}, \vec{a}, \vec{y}\,'_i) \wedge C(\vec{v})[\vec{y}_i / \vec{y}\,'_i]. \tag{5.1}$$

Thus, a complete partitioned preimage is found by computing $\textsc{PreImgSA}_i(C)$ for each subrelation in turn and merging partitions with identical h-value.

5.2.1 Guided Weak Precomponents

The main computation of the guided weak precomponent is to find the preimage of a set of states S and prune it for SAs where the state is in C

$$\textsc{PreCompW}_i(C, S) \;\equiv\; \textsc{PreImgSA}_i(S) \setminus C \times Act. \tag{5.2}$$

The algorithm computing the guided weak precomponent is shown in Figure 5.2. The input

function GPRECOMPW($\mathbf{C}$)
$Q \leftarrow emptyQueue$;
for $j = 1$ **to** $|\mathbf{C}|$
  for $i = 1$ **to** n
    $wSA \leftarrow \textsc{PreCompW}_i(C, \mathbf{C}[h_j])$
    $Q \leftarrow \textsc{Insert}(Q, \langle wSA, h_j - \delta h_i \rangle)$
if $|Q| = 0$ **then return** $emptyMap$
else return $\textsc{RemoveTop}(Q)$

Figure 5.2: The guided weak precomponent.

to the function is a map of covered states $\mathbf{C}$ where each entry contains a set of states with identical h-value. A priority queue Q stores the weak precomponents of $\mathbf{C}$. The keys of Q is the h-values of the set of SAs forming the entries of Q. As usual, the set of keys of Q are sorted ascendingly and a node inserted in Q is merged with any existing node with identical h-value. $\textsc{RemoveTop}(Q)$ returns a map with the top of Q as its only element.

Let GUIDEDWEAK denote the GNDP algorithm using the guided weak precomponent function. It can be shown that GUIDEDWEAK is sound, complete and terminating. However, since a pure heuristic search strategy is employed, solutions are not guaranteed to be weak distance optimal like solutions computed with the WEAK algorithm.

Theorem 5.2 (Correctness of GuidedWeak) *The* GUIDEDWEAK *planning algorithm is correct. The algorithm returns "no solution exists" iff no solution exists, otherwise it returns a valid solution.*

Proof. This follows from the soundness, completeness, and termination theorems of GUIDEDWEAK proven in Appendix B. □

5.2.2 Guided Strong Precomponents

The main computation of the guided strong precomponent is to find a preimage of a set of states S and prune it for SAs where the state either is in C or the SA can lead outside of C

$$\textsc{PreCompS}_i(C,S) \;\equiv\; \Big(\textsc{PreImgSA}_i(S) \setminus \textsc{PreImgSA}(\overline{C})\Big) \setminus C \times Act. \quad (5.3)$$

The algorithm computing the guided strong precomponent is shown in Figure 5.3. It is similar to GPRECOMPS except that the function $\textsc{PreCompS}_i$ is used instead of $\textsc{PreCompW}_i$.

function GPRECOMPS$(\mathbf{C})$
$Q \leftarrow emptyQueue$;
for $j = 1$ **to** $|\mathbf{C}|$
  for $i = 1$ **to** n
    $sSA \leftarrow \textsc{PreCompS}_i(C, \mathbf{C}[h_j])$
    $Q \leftarrow \textsc{Insert}(Q, \langle sSA, h_j - \delta h_i \rangle)$
if $|Q| = 0$ **then return** $emptyMap$
else return $\textsc{RemoveTop}(Q)$

Figure 5.3: The guided strong precomponent.

Let GUIDEDSTRONG denote the GNDP algorithm using the guided strong precomponent function. It can be shown that GUIDEDSTRONG is sound, complete and terminating. However, similarly to GUIDEDWEAK, since a pure heuristic search strategy is employed, solutions are not guaranteed to be strong distance optimal like solutions computed with the STRONG algorithm.

Theorem 5.3 (Correctness of GuidedStrong) *The* GUIDEDSTRONG *planning algorithm is correct. The algorithm returns "no solution exists" iff no solution exists, otherwise it returns a valid solution.*

Proof. This follows from the soundness, completeness, and termination theorems of GUIDEDSTRONG proven in Appendix B. □

5.2.3 Guided Strong Cyclic Precomponents

The guided strong cyclic precomponent is fairly different from the weak and strong precomponents. At each call, the algorithm builds a set of candidate SAs from a search tree of weak precomponents grown from the set of covered states C. For each extension of the candidate set, the SCPLANAUX function defined in Section 3.2.2 is called to extract a strong cyclic precomponent from the candidate, if possible. The search queue Q stores the

```
function GPRECOMPSC(C)
  Q ← emptyQueue
  for j = 1 to |C|
    for i = 1 to n
      cSA ← PRECOMPW_i(C, C[h_j])
      Q ← INSERT(Q, ⟨cSA, 1, h_j − δh_i⟩)
  wSA ← ∅; wS ← emptyMap
  repeat
    if |Q| = 0 then return emptyMap
    ⟨pSA, d, h⟩ ← REMOVETOP(Q)
    pSA ← pSA \ wSA
    if pSA ≠ ∅ then
      pS ← STATES(pSA)
      wS[h] ← wS[h] ∪ pS
      for i = 1 to n
        cSA ← PRECOMPW_i(C, pS)
        Q ← INSERT(Q, ⟨cSA, d + 1, h − δh_i⟩)
      wSA ← wSA ∪ pSA
      scSA ← SCPLANAUX(wSA, C)
  until scSA ≠ ∅
  P_c ← emptyMap
  for k = 1 to |wS|
    P_c[h_k] ← wS[h_k] ∩ scSA
  return P_c
```

Figure 5.4: The guided strong cyclic precomponent.

frontier nodes of a search tree of weak precomponents generated from the states C in the current plan. Each node is associated with its h-value as usual, however, for this algorithm, we also associate a node with its depth d in the search tree. In addition, the highest priority is given to nodes with smallest sum of h and d. In case of a tie, the highest priority is given to the node with smallest depth. When inserting a new node in Q, it will only be merged with an existing node in Q if this node has identical h and d value. For each call to GPRECOMPSC, a new search tree is generated. First, the weak precomponents of the states C covered by the current plan are added to Q (l.1-5). These are all at depth 1 in the tree. Then, the candidate SAs wSA for the strong cyclic precomponent and auxiliary variables are initialized (l.6). The repeat loop (l.7-19) performs a guided version of the expansion and pruning of wSA carried out by PRECOMPSC(C). The effect of taking the depth in the search tree into account is that the set of candidate SAs does not form a narrow beam in the state space that is unlikely to contain a strong cyclic precomponent. When a non-empty strong cyclic precomponent $scSA$ is found, the SAs of this precomponent are partitioned with respect to their h-value (l.20-22) and the resulting map is returned.

Let GUIDEDSTRONGCYCLIC denote the GNDP algorithm using the strong cyclic precomponent function. It can be shown that GUIDEDSTRONGCYCLIC is sound, complete and terminating.

Theorem 5.4 (Correctness of GuidedStrongCyclic) *The* GUIDEDSTRONGCYCLIC *planning algorithm is correct. The algorithm returns "no solution exists" iff no solution exists, otherwise it returns a valid solution.*

Proof. This follows from the soundness, completeness, and termination theorems of GUIDEDSTRONGCYCLIC proven in Appendix B. □

5.3 Experimental Results

The performance of the guided non-deterministic planning algorithms has been evaluated in three non-deterministic domains and two deterministic domains. We include the deterministic domains since only a limited number of parameterized non-deterministic domains with efficient search heuristics has been modeled so far. The deterministic domains may provide some information about robustness of the search approach of non-deterministic state-set branching to different heuristics.

All experiments are carried out using the BIFROST 0.7 search engine and the experimental setting described in Appendix A. As usual, n denotes the number of BDD-nodes allocated to represent the shared BDD, and c denotes the number of BDD nodes allocated to represent BDDs in the operator caches used to implement dynamic programming. Total CPU time includes time spent on allocating memory of the BDD package, parsing the problem description and, in case of PDDL problems, analysing the problem in order to make a compact Boolean state encoding. Time out changes between the experiments. The algorithms are out of memory when they start page faulting to the hard drive at approximately 450 MB RAM.

5.3.1 Non-Deterministic Domains

The three non-deterministic domains are a non-deterministic version of the 8-Puzzle, a real-world steel producing plant of SIDMAR in Ghent, Belgium used as an ESPRIT case study [56], and a real-world domain for Power Supply Restoration (PSR) introduced in [162].

Non-Deterministic 8-Puzzle

To make the 8-Puzzle domain non-deterministic, we assume that up and down moves of the blank space may move left and right as well, as shown in Figure 5.5. Left and right moves are deterministic in order to ensure that a strong plan exists for any reachable initial state. However, we only consider actions that at most reduce the distance to the initial state by one. Otherwise, the sum of Manhattan distances becomes a too conservative heuristic.

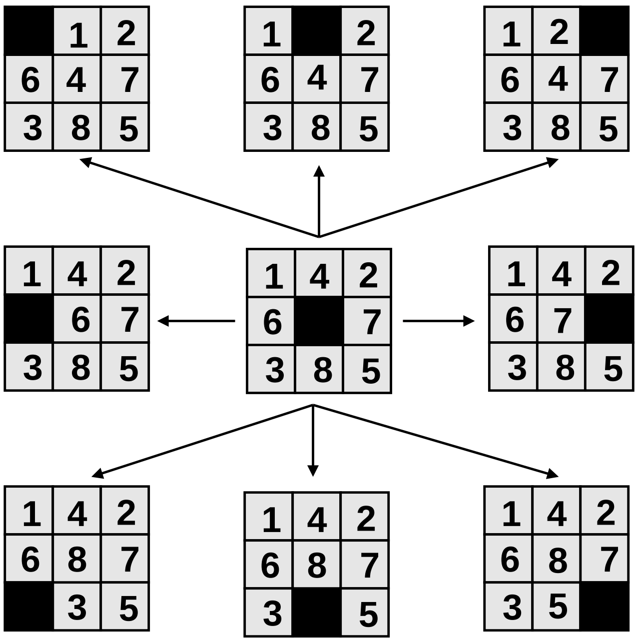


Figure 5.5: Moves of the blank space and their possible outcomes.

We consider problems where the minimum length of a path from the initial state to the goal state grows linearly from 8 to 23. The BDD package was initialized with $n = 4M$ and $c = 700K$. The threshold for merging partitions in the disjunctive transition relation partitioning was 5000. Memory allocation and transition relation construction took 1.56 and 1.34 seconds respectively for all experiments. The results are shown in Figure 5.6. Each data point shown in the graphs is the average of 3 computational results. The results show a dramatic positive impact of guiding the search for all three algorithms. As depicted in the graphs, it may reduce not only the total computation time but also the size of the produced plans. This may be somewhat surprising since the guided algorithms apparently repeat a large number of computations. The previous results of such recomputations, however, may often be stored in the operator cache of the BDD package and may therefore not cause a significant computation overhead.

SIDMAR

The SIDMAR domain is an abstract model of a real-world steel producing plant in Ghent, Belgium used as an ESPRIT case study [56]. The layout of the steel plant is shown in Figure 5.7. The goal is to cast steel of different qualities. Pig iron is poured portion-wise in ladles by the two converter vessels. The ladles can move autonomously on the two east-west tracks. However, two ladles can not pass each other and there can at most be one ladle between machines. Ladles are moved in the north-south direction by the two overhead cranes. The pig iron must be treated differently to obtain steel of different qualities. Before empty ladles are moved to the storage place, the steel is cast by the continuous casting machine. A ladle can only leave the casting machine if there already is a filled ladle at the holding place. The actions of machine 1,2,4, and 5 are non-deterministic. They may either cause the steel in the ladles to be treated or the machine to break. To ensure that a strong plan exists, actions have been added to the domain that can fix failed machines.

We consider producing steel from two ladles. They both need an initial treatment on machine 1 or 4 and 2 or 5. One of the ladles in addition needs a treatment on machine 3 and a final treatment on machine 2 or 5 before being cast. Non-determinism is caused by machine failures. We consider 6 problems where the goal states correspond to situations with growing distances from the initial state during the production of these two ladles. The number of completed treatments is used as heuristic function. Notice that this heuristic is relatively weak compared to the sum of Manhattan distances used for the 8-Puzzle since it severely underestimates the minimum distance to the initial state. The BDD package was initialized with $n = 8M$ and $c = 700K$. The threshold for merging partitions in the disjunctive transition relation partitioning was 5000. Memory allocation and transition relation construction took 2.34 and 0.22 seconds respectively for all experiments. The results are shown in Figure 5.8. Again, we observe a large performance gain obtained by

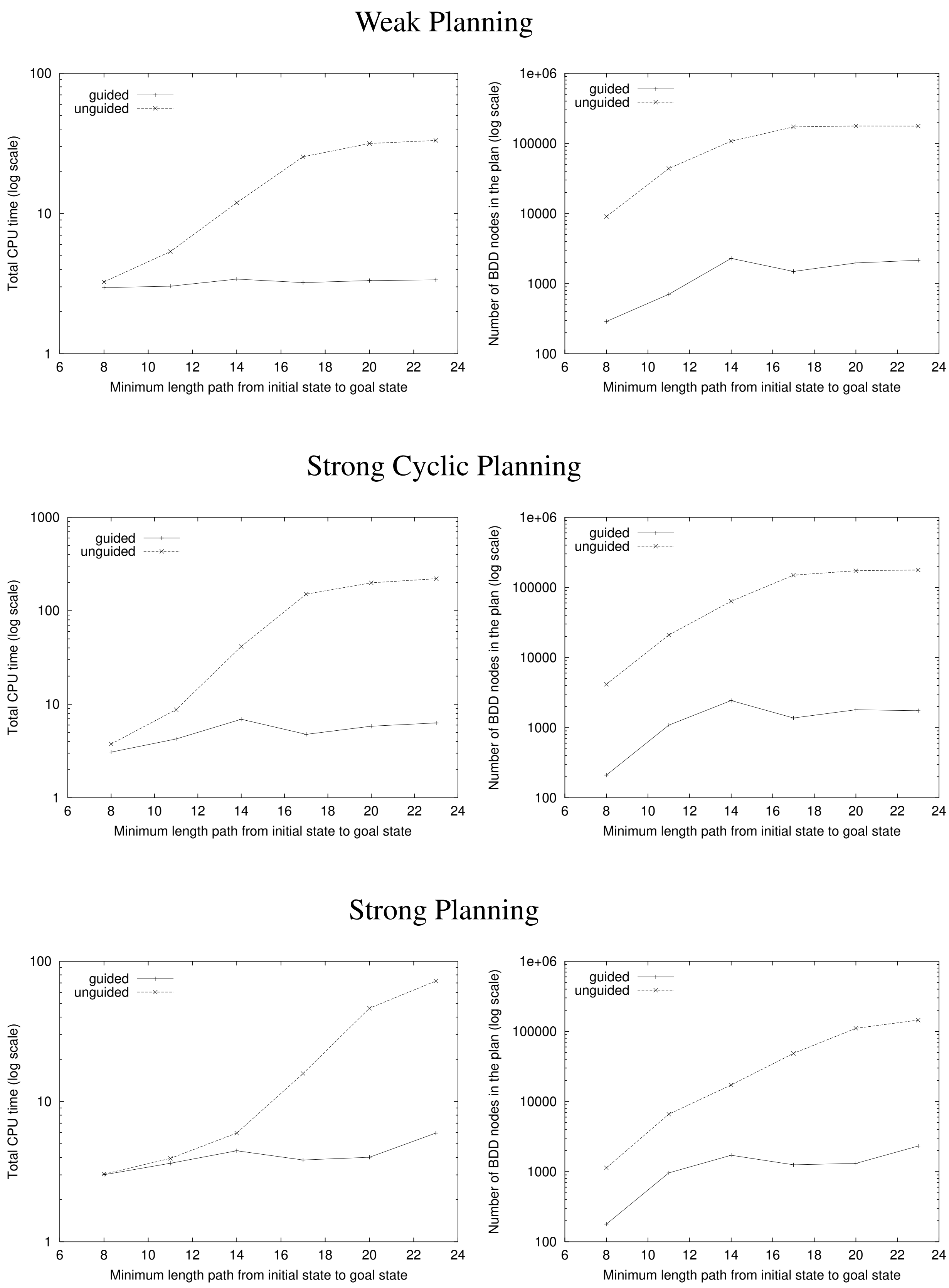


Figure 5.6: Results of the non-deterministic 8-Puzzle experiments.

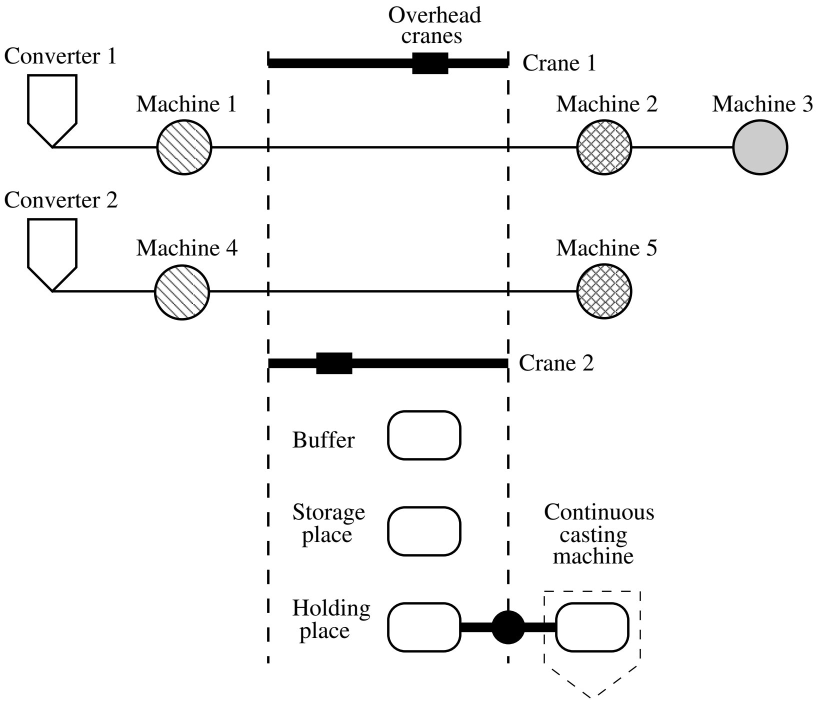


Figure 5.7: Layout of the SIDMAR steel plant.

guiding the search for all three algorithms both in terms of total CPU time and the plan size. These results are encouraging since SIDMAR is a real-world domain and non-determinism is caused by realistic faults. In addition, the results demonstrate that even a very weak heuristic may have a substantial positive effect on performance.

PSR

The Power Supply Restoration domain (PSR) is a network of electric lines connected via switching devices (SDs), and fed via circuit-breakers (CBs). Switching devices and circuit breakers can either be open or closed. A circuit-breaker supplies power when it is closed, and a switching device stops the power propagation if it is open. Consumers may be located on any line and are supplied only when the line is supplied. We assume that each closed circuit-breaker forms a feeder. A feeder is a tree consisting of closed switching devices and lines reachable downstream from the circuit breaker. The leafs are open switching devices and dead end lines.

Example 5.1 The "simple" PSR domain investigated in [14] is shown in Figure 5.9. In the depicted configuration, it only has a single feeder rooted in CB_2. ◊

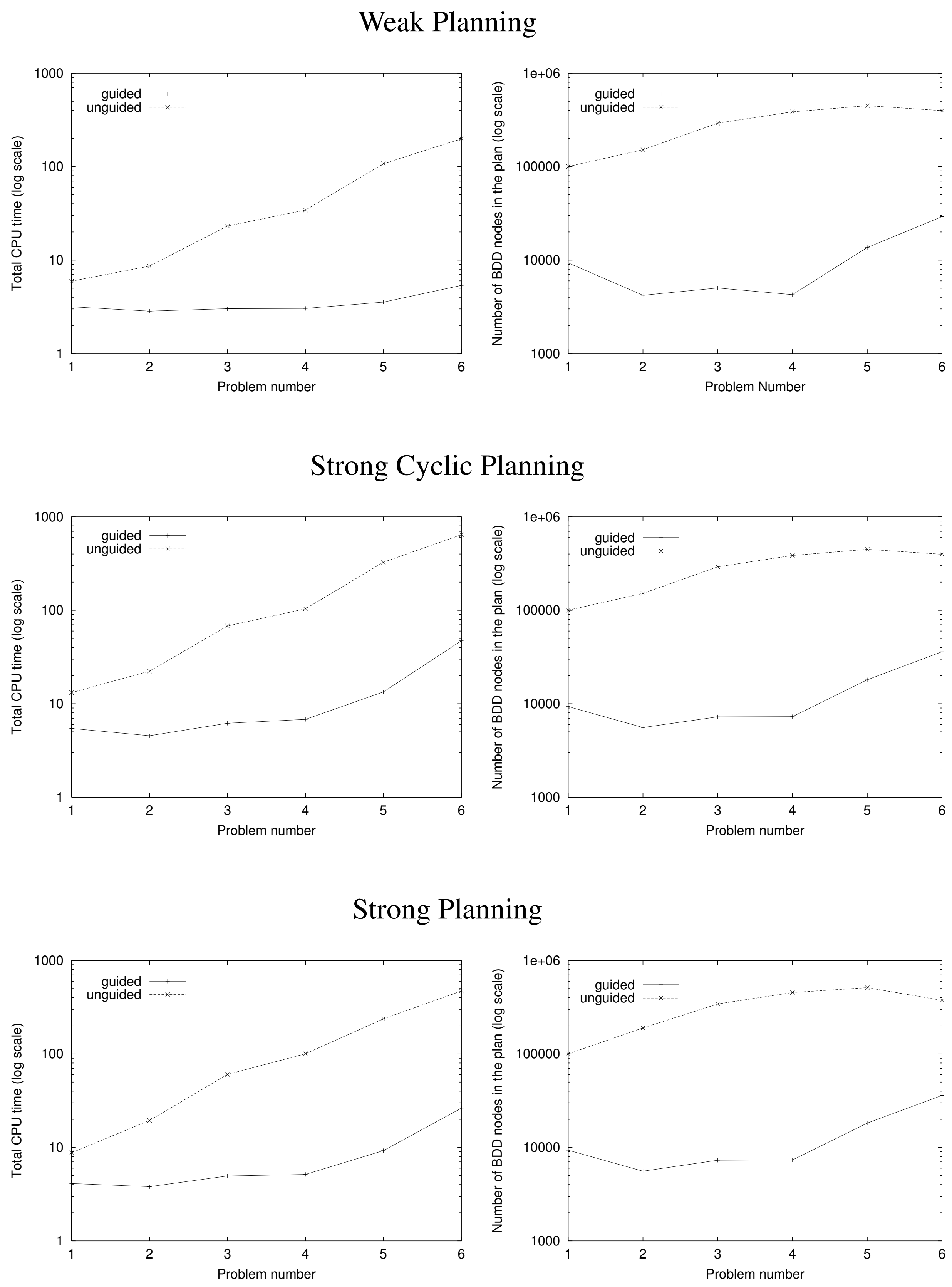


Figure 5.8: Results of the SIDMAR experiments.

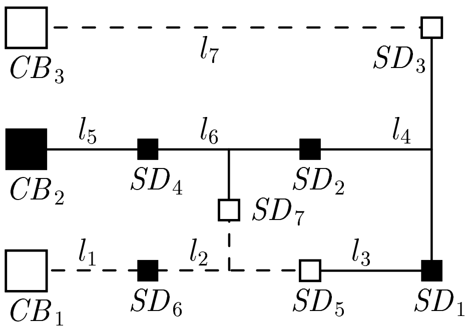


Figure 5.9: The "simple" PSR domain studied in [14]. A filled box denote that the associated circuit-breaker or switching device is closed. Supplied and unsupplied lines are drawn solid and dashed, respectively.

In the original definition of PSR domains, each unit in the system may fail. Lines may short circuit, and switches may get stuck in one of their two positions. In addition, states are assumed only to be partially observable. We consider a simplified version of the domain where states are fully observable and lines do not fail. The actions of the simplified domain is to open and close switching devises and circuit breakers. The actions are non-deterministic, they may open and close these units correspondingly or cause the units to break permanently and get stuck in their current position.

The studied networks are on the linear form shown in Figure 5.10 with n ranging from 5 to 35. Initially, every unit is open and the goal is to feed each line. Since any combination of errors may happen, neither a strong nor strong cyclic solution exists. The heuristic used to guide the search is the number of lines with power. The results of the experiment is shown

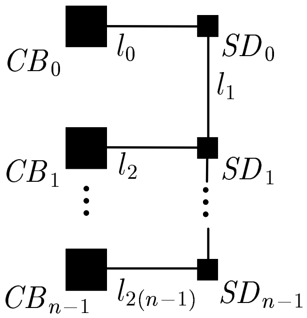


Figure 5.10: The linear PSR networks used for experiments.

in Figure 5.11 The BDD package was initialized with $n = 15M$ and $c = 500K$. The threshold for merging partitions in the disjunctive transition relation partitioning was 5000. Memory allocation took 3.41 seconds for all experiments. Transition relation construction took between 0.05 and 1.21 seconds. Again, we see a substantial positive impact of guiding

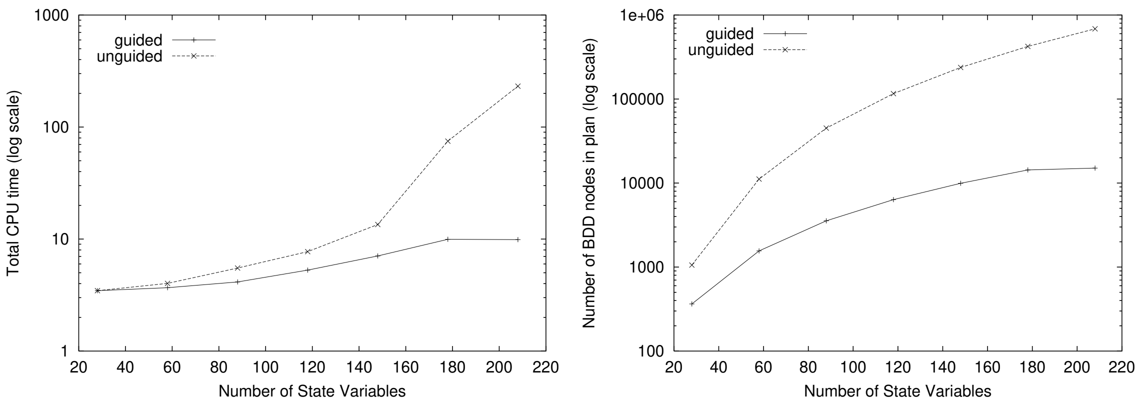


Figure 5.11: Results of the guided weak algorithm on the linear PSR problems.

the search.

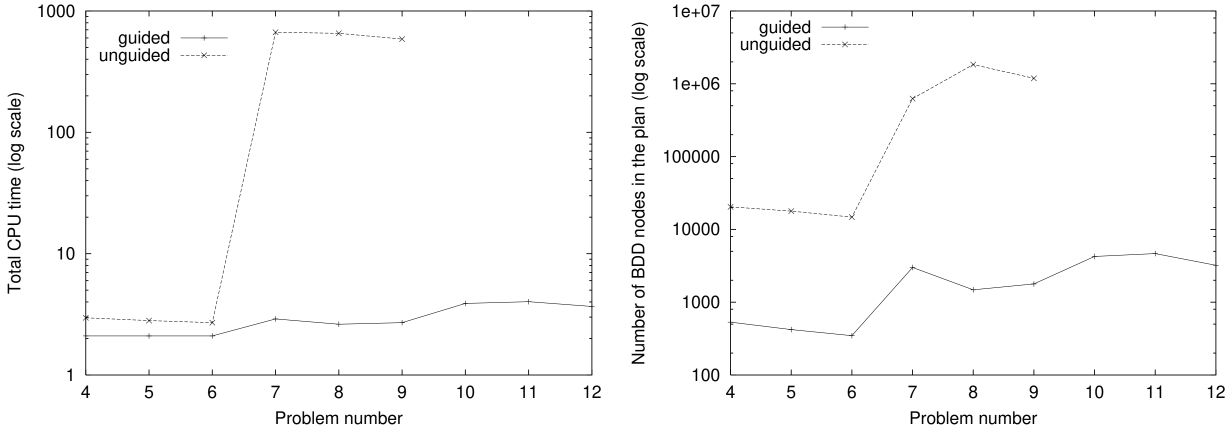
5.3.2 Deterministic Domains

The two deterministic domains are the logistics domain described in Section 4.3.2 and the ZenoTravel domain described in Section 4.3.2.

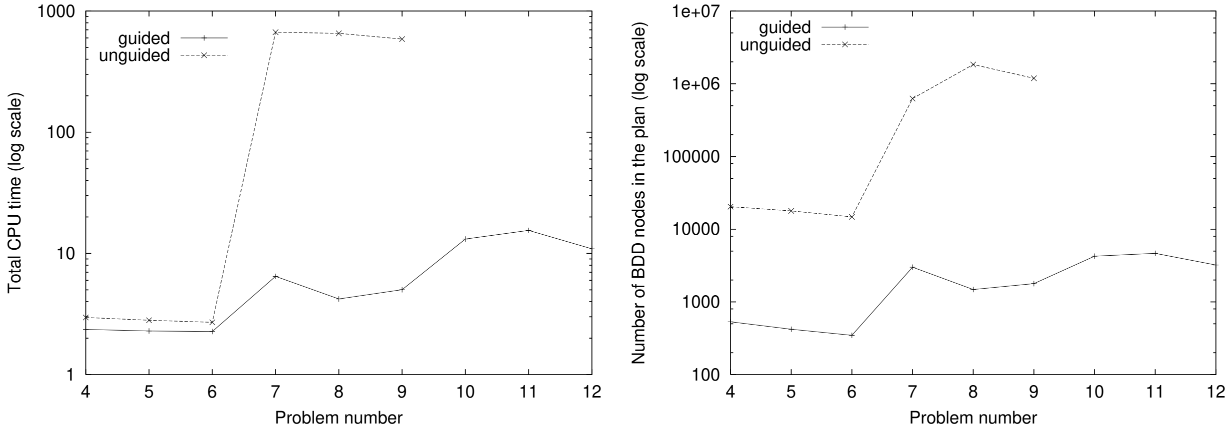
Logistics

For the logistics domain, we again study the problems from the STRIPS track of the AIPS 2000 planning competition and use the HSPr heuristic to guide the search (recall that this is a heuristic for backward search). The results of the experiment is shown in Figure 5.12. The BDD package was initialized with $n = 12M$ and $c = 400K$. The threshold for merging partitions in the disjunctive transition relation partitioning was 5000. Memory allocation took 1.9 seconds for all experiments. Transition relation construction took between 0.10 and 0.77 seconds. The results show a significant performance improvement for each algorithm. However, since the domain is deterministic the blind weak, strong cyclic and strong precomponents are identical. Thus, we may expect the guided precomponents and the solutions returned by the guided weak, strong cyclic and strong algorithm to be identical. This seems to be the case. Even though each algorithm computes fairly similar solutions, the results bring further evidence that the general guiding strategy employed by the algorithms has good performance for a wide range of heuristics. In addition, the results show that computational overhead of the algorithms is fairly similar. Not surprisingly, the weak algorithm seems to have the smallest overhead.

Weak Planning



Strong Cyclic Planning



Strong Planning

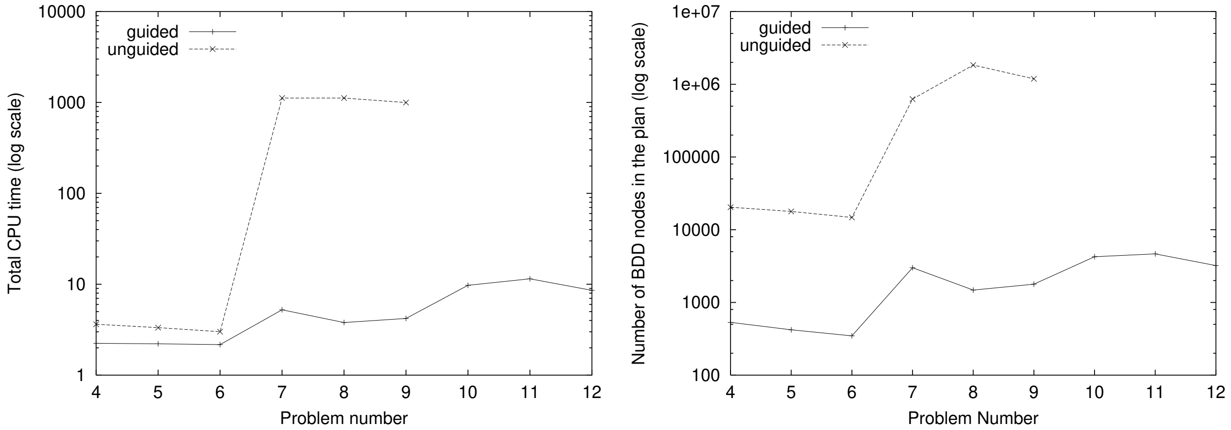
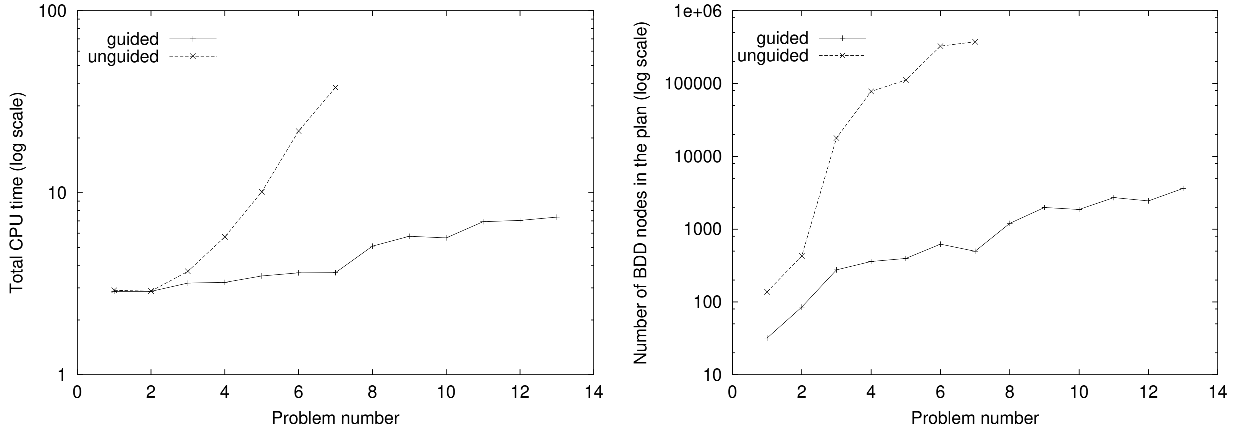
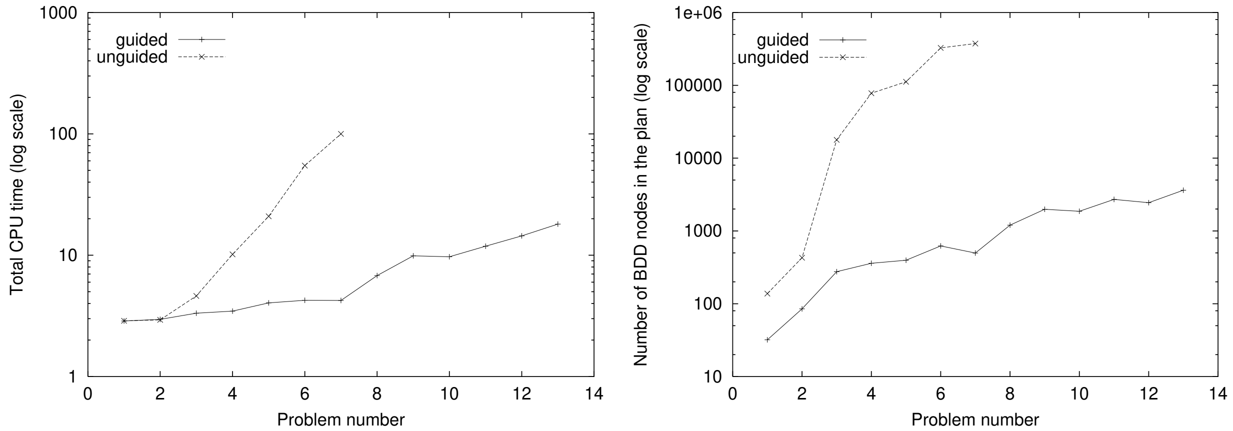


Figure 5.12: Results of the Logistics experiments.

Weak Planning



Strong Cyclic Planning



Strong Planning

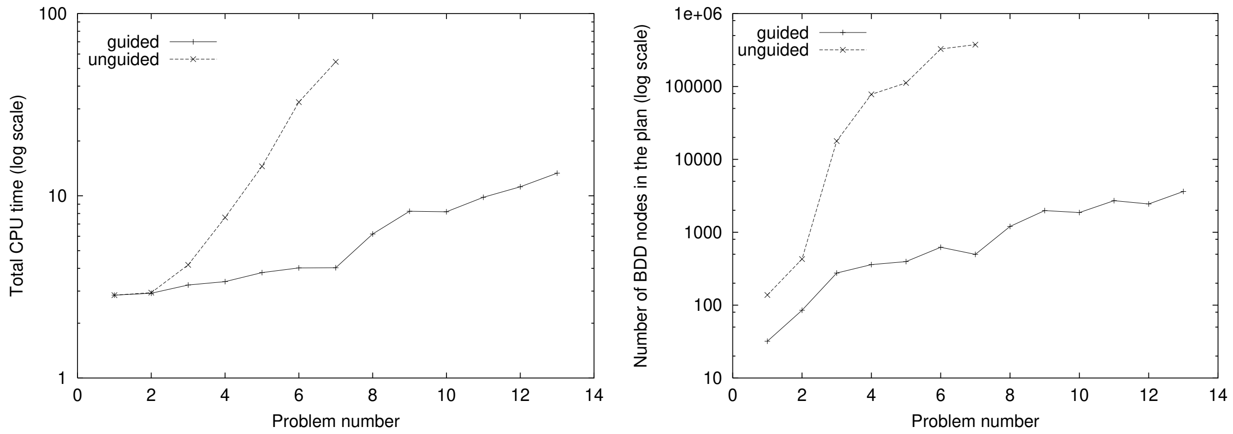


Figure 5.13: Results of the ZenoTravel experiments.

Zeno Travel

For the ZenoTravel domain, we again study the problems from the STRIPS track of the AIPS 2002 planning competition and use the HSPr heuristic to guide the search. The results of the experiment is shown in Figure 5.13 The BDD package was initialized with $n = 12M$ and $c = 400K$. The threshold for merging partitions in the disjunctive transition relation partitioning was 5000. Memory allocation took 2.70 seconds for all experiments. Transition relation construction took between 0.11 and 18.03 seconds. The results and their interpretation are similar to the Logistics domain.

5.4 Conclusion

Our investigation of non-deterministic state-set branching has shown that it is possible to employ branching partitionings in non-deterministic domains and define pure heuristic search strategies of the weak, strong cyclic, and strong planning algorithms. The experimental results show that non-deterministic state-set branching can lead to large performance gains compared to blind search, not only in terms of CPU time but also in terms of plan size. The main limitation of the approach is that no optimality guarantees can be given for weak and strong solutions. Another limitation is that the algorithms in order to provide completeness perform a large number of recomputations when building a complete breadth-first search frontier in each iteration. Such recomputations may be avoided due to the extensive caching of previous results by the BDD package. However, we cannot rely on unstructured caching for memory intense problems. An interesting direction for future work is to define a method for *maintaining* a complete search frontier instead of simply recomputing it.

5.5 Summary

In this chapter, we have seen how the core ideas of state-set branching can be applied to non-deterministic planning. Pure heuristic versions of the weak, strong cyclic, and strong algorithms have been developed and formally proven to be correct. A range of experimental results in three non-deterministic and two deterministic domains using four different heuristics show that the new guided algorithms may have dramatically better performance than the ordinary blind algorithms both in terms of search time and solution size.

Chapter 6

Fault Tolerant Planning

In the previous two chapters, the focus has been on lowering the complexity of BDD-based deterministic and non-deterministic search. In this, and the following chapter, we will shift focus and introduce extensions to the non-deterministic domain model in order to improve the quality of the produced solutions. A key observation is that non-determinism in the real world often is caused by infrequent errors that make otherwise deterministic actions fail. In many cases, no actions can be guaranteed to succeed. For such domains, it may be hard or even impossible to generate plans that can recover from any combination of errors. In this chapter, we propose a new framework called *fault tolerant planning* [96] to handle this kind of non-determinism.

Section 6.1 defines fault tolerant planning domains and n-fault tolerant plans that are robust to n failures occurring during the execution of the plan. In Section 6.2, we show how optimal n-fault tolerant plans can be generated with the strong algorithm. Due to non-local error states it turns out, however, to be hard to guide the search efficiently with non-deterministic state-set branching when using this approach. We therefore develop a specialized guided 1-fault tolerant planning algorithm 1-GFTP that decouples the guiding toward error states and guiding toward the initial state. In Section 6.3, we present a range of experimental results that show that the specialized algorithm indeed may be necessary for solving real-world problems efficiently. Finally, Section 6.4 draws conclusions and discusses directions for future work.

6.1 N-Fault Tolerant Planning Problems

Fault tolerant planning assumes that actions have primary and secondary effects. The primary effect models the usual deterministic behavior of the action, while the secondary effect models the error effects. N-Fault tolerant plans are robust to n errors or faults occurring during the execution of the plan. This definition of fault tolerance is closely connected to fault tolerance concepts in control theory and engineering. Every time we board a two engined aircraft, we enter a 1-fault tolerant system: a single engine failure is recoverable, but two engines failing may lead to an unrecoverable breakdown of the system.

An n-fault tolerant plan is not as restrictive as a strong plan that requires that the goal can be reached in a finite number of steps independent of the number of errors. In many cases, a strong plan does not exist because all possible errors must be taken into account. This is not the case for fault tolerant plans, and if errors are infrequent, they may still be very likely to succeed. A fault tolerant plan is also not as restrictive as a strong cyclic plan. An execution of a strong cyclic plan will never reach states not covered by the plan unless it is a goal state. Thus, strong cyclic plans also have to take all error combinations into account. Weak plans, on the other hand, are more relaxed than fault tolerant plans. Fault tolerant plans, however, are almost always preferable to weak plans because they give no guarantees for *all* the possible outcomes of actions. For fault tolerant plans, any action may fail, but only a limited number of failures are recoverable.

A fault tolerant planning domain is similar to a deterministic planning domain. However, in addition to the primary effect of actions, we add a secondary effect that describes the outcome of a failure. Since an action can often fail in many different ways, we allow the secondary effect to lead to one of several possible next states. Thus, secondary effects are non-deterministic.

Definition 6.1 (Fault Tolerant Planning Domain) *A fault tolerant planning domain is a tuple $\langle S, Act, \rightarrow, \rightsquigarrow \rangle$ where S is a finite set of states, Act is a finite set of actions, $\rightarrow \subseteq S \times Act \times S$ is a deterministic transition relation of primary effects, and $\rightsquigarrow \subseteq S \times Act \times S$ is a non-deterministic transition relation of secondary effects. Instead of $(s, a, s') \in \rightarrow$ and $(s, a, s') \in \rightsquigarrow$, we write $s \xrightarrow{a} s'$ and $s \overset{a}{\rightsquigarrow} s'$, respectively.*

The planning language NADL$^+$ described in Appendix A may be used to represent fault tolerant planning domains. An n-fault tolerant planning problem is a deterministic planning problem extended with the fault limit n.

Definition 6.2 (N-Fault Tolerant Planning Problem) *An n-fault tolerant planning problem is a tuple $\langle \mathcal{D}, s_0, G, n \rangle$ where $\mathcal{D}$ is a fault tolerant planning domain, $s_0 \in S$ is an initial state, $G \subseteq S$ is a set of goal states, and $n : \mathbb{N}$ is an upper bound on the number of faults the plan must be able to recover from.*

An n-fault tolerant plan is defined via a transformation of an n-fault tolerant planning problem to a non-deterministic planning problem. The transformation adds a fault counter f to the state description and models secondary effects only when $f \leq n$.

Definition 6.3 (Induced Non-Deterministic Planning Problem) *Let $\mathcal{P} = \langle \mathcal{D}, s_0, G, n \rangle$ where $\mathcal{D} = \langle S, Act, \rightarrow, \rightsquigarrow \rangle$ be an n-fault tolerant planning problem. The non-deterministic planning problem induced from $\mathcal{P}$ is $\mathcal{P}^{nd} = \langle \mathcal{D}^{nd}, \langle s_0, 0 \rangle, G \times \{0, \cdots, n\} \rangle$ where $\mathcal{D}^{nd} = \langle S^{nd}, Act^{nd}, \rightarrow^{nd} \rangle$ and is given by*

- $S^{nd} = S \times \{0, \cdots, n\}$,
- $Act^{nd} = Act$,
- $\langle s, f \rangle \stackrel{a}{\rightarrow}^{nd} \langle s', f' \rangle$ *iff*
  - $s \stackrel{a}{\rightarrow} s'$ *and* $f' = f$, *or*
  - $s \stackrel{a}{\rightsquigarrow} s'$, $f < n$, *and* $f' = f + 1$.

Definition 6.4 (Valid N-Fault Tolerant Plan) *A valid n-fault tolerant plan for the n-fault tolerant planing problem $\mathcal{P}$ is a non-deterministic plan π for the non-deterministic planning problem induced from $\mathcal{P}$ where $\mathcal{M}(\pi), s_0^{nd} \models \mathtt{AF}\, G^{nd}$.*

Thus, an n-fault tolerant plan is valid if any execution path where at most n failures happen eventually reaches a goal state. An n-fault tolerant plan is optimal if it has minimum worst case execution length.

Definition 6.5 (Optimal N-Fault Tolerant Plan) *An optimal n-fault tolerant plan is a valid n-fault tolerant plan π where* $\textsc{Max}(s_0^{nd}, G^{nd}, \pi) = \textsc{SDist}(s_0^{nd}, G^{nd})$

6.2 N-Fault Tolerant Planning Algorithms

One might suggest using a deterministic planning algorithm to generate n-fault tolerant plans. Consider for instance synthesizing a 1-fault tolerant plan in a domain where there is a non-faulting plan of length k and at most f error states of any action. It is tempting to claim that a 1-fault tolerant plan then can be found using at most kf calls to a classical deterministic planning algorithm. This analysis, however, is flawed. It only holds for evaluating a given 1-fault tolerant plan. It neglects that many additional calls to the classical planning algorithm may be necessary in order to *find* a valid solution. Instead, we need an efficient approach for finding plans for many states simultaneously. This can be done with BDD-based non-deterministic planning. We first observe that it follows directly from Definition 3.9 that the strong algorithm returns a valid n-fault tolerant plan, if it exists, when given the induced non-deterministic planning problem as input. Moreover, if the blind strong algorithm is used to generate the solution, it follows from Theorem 3.3 that the returned n-fault tolerant plan is optimal. Let n-FTP$_S$ denote the STRONG algorithm applied to an n-fault tolerant planning problem. Since the performance of blind strong planning is limited, we also consider solving n-fault tolerant planning problems with the guided version of strong planning defined in previous chapter. Let n-GFTP$_S$ denote the GUIDEDSTRONG algorithm applied to an n-fault tolerant planning problem.

We may expect n-GFTP$_S$ to be efficient when secondary effects are local in the state space because they then will be covered by the search beam of n-GFTP$_S$. In practice, however, secondary effects may be permanent malfunctions that due to their impact on the domain cause a transition to a non-local state. That is a state from which no short path of primary effects exists to the source state. Indeed, in theory, the location of secondary effects may be completely uncorrelated with the location of primary effects. To solve this problem, we develop a specialized algorithm where the planning for primary and secondary effects is decoupled. We constrain our investigation to 1-fault tolerant planning and introduce two algorithms: 1-FTP using blind search and 1-GFTP using guided search. The input to these algorithms is a 1-fault tolerant planning problem, not its induced non-deterministic planning problem.

The 1-FTP algorithm is shown in Figure 6.1. The function PREIMGSA$_f$ computes the

function 1-FTP(s_0, G)
  $F^0 \leftarrow \emptyset; C^0 \leftarrow G$
  $F^1 \leftarrow \emptyset; C^1 \leftarrow G$
  while $s_0 \notin C^0$
    $f^0_c \leftarrow \text{PREIMGSA}(C^0) \setminus C^0 \times Act$
    $f^0 \leftarrow f^0_c \setminus \text{PREIMGSA}_f(\overline{C^1})$
    while $f^0 = \emptyset$
      $f^1 \leftarrow \text{PREIMGSA}(C^1) \setminus C^1 \times Act$
      if $f^1 = \emptyset$ **then return** "no solution exists"
      $F^1 \leftarrow F^1 \cup f^1$
      $C^1 \leftarrow C^1 \cup \text{STATES}(f^1)$
      $f^0 \leftarrow f^0_c \setminus \text{PREIMGSA}_f(\overline{C^1})$
    $F^0 \leftarrow F^0 \cup f^0$
    $C^0 \leftarrow C^0 \cup \text{STATES}(f^0)$
  return $\langle F^0, F^1 \rangle$

Figure 6.1: The 1-FTP algorithm.

preimage of secondary effects. 1-FTP returns two non-deterministic plans F^0 and F^1 for the fault tolerant domain, where F^0 is robust to one fault while F^1 is a recovery plan.

Example 6.1 An example of the non-deterministic plans F^0 and F^1 returned by 1-FTP is shown in Figure 6.2 $\Diamond$

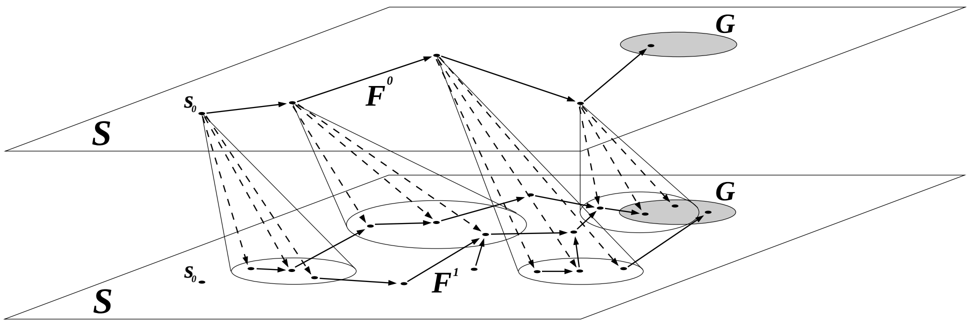


Figure 6.2: An example of the non-deterministic plans F^0 and F^1 returned by 1-FTP. Primary and secondary effects of actions are drawn with solid and dashed lines, respectively. In this example, we assume that F^0 forms a sequence of actions from the initial state to a goal state, while F^1 recovers all the possible faults of actions in F^0.

1-FTP performs a backward search from the goal states that alternate between blindly expanding F^0 and F^1 such that failure states of F^0 always can be recovered by F^1. Initially F^0 and F^1 are assigned to empty plans (l. 1-2). The variables C^0 and C^1 are states covered by the current plans in F^0 and F^1. They are initialized to the goal states since these states are covered by zero length plans. In each iteration of the outer loop (l. 3-13), F^0 is expanded with SAs in f^0 (l. 12-13). First, a candidate f^0_c is computed. It is the preimage of the states in F^0 pruned for SAs of states already covered by F^0 (l. 4). The variable f^0 is assigned to f^0_c restricted to SAs for which all error states are covered by the current recovery plan (l. 5). If f^0 is empty the recovery plan is expanded in the inner loop until f^0 is nonempty (l. 6-11). If the recovery plan at some point has reached a fixed point and f^0 is still empty, the algorithm terminates with failure, since in this case, no recovery plan exists (l. 8). We claim without proof that 1-FTP is *sound*, *complete*, and *terminating*.

1-FTP expands both F^0 and F^1 blindly. An inherent strategy of the algorithm, though, is not to expand F^1 more than necessary to recover the faults of F^0. This is not the case for n-FTP$_S$ that does not distinguish states with different number of faults. The aggressive strategy of 1-FTP, however, makes it suboptimal as the example in Figure 6.3 shows. In the first two iterations of the outer loop, $\langle p_2, b\rangle$ and $\langle p_1, b\rangle$ are added to F^0 and nothing is added to F^1. In the third iteration of the outer loop, F^1 is extended with $\langle p_2, b\rangle$ and $\langle q_2, a\rangle$ and F^0 is extended with $\langle q_2, a\rangle$. In the last two iterations of the outer loop, $\langle q_1, a\rangle$ and $\langle s_0, a\rangle$ are added to F^0. The resulting plan is

$$F^0 \quad = \quad \{\langle s_0, a\rangle, \langle q_1, a\rangle, \langle q_2, a\rangle, \langle p_1, b\rangle, \langle p_2, b\rangle\}$$

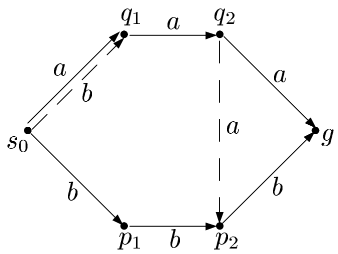


Figure 6.3: A problem with a single goal state g showing that 1-FTP may return suboptimal solutions. Dashed lines indicate secondary effects. Notice that action a and b only have secondary effects in q_2 and s_0, respectively. In all other states, the actions are assumed always to succeed.

$$F^1 = \{\langle p_2, b\rangle, \langle q_2, a\rangle\}.$$

The worst case length of this 1-fault tolerant plan is 4. However, a 1-fault tolerant plan

$$\begin{aligned} F^0 &= \{\langle s_0, b\rangle, \langle p_1, b\rangle, \langle p_2, b\rangle\} \\ F^1 &= \{\langle q_1, a\rangle, \langle q_2, a\rangle\} \end{aligned}$$

with worst case length of 3 exists.

Despite the different search strategies applied by 1-FTP and 1-FTP$_S$, they both perform blind search. A more interesting algorithm is a guided version of 1-FTP called 1-GFTP based on the non-deterministic state-set branching framework introduced in previous chapter. The over all design goal of 1-GFTP is to guide the expansion of F^0 toward the initial state and guide the expansion of F^1 toward the failure states of F^0. However, this can be accomplished in many different ways. Below we evaluate three different strategies. For each algorithm, F^0 is guided in a pure heuristic manner toward the initial state using the approach employed by n-GFTP$_S$.

The first strategy is to assume that failure states are local and guide F^1 toward the initial state as well. The resulting algorithm is similar to 1-GFTP$_S$ and has poor performance. The problem is that the pure heuristic approach causes F^1 only to cover a narrow beam of states in the state space. Error states not within close distance to the primary effects tend not to be covered by F^1. The strategy can be improved by widening the beam by taking the search depth into account. However, this does not provide a satisfactory solution for non-local states.

The second strategy is ideal in the sense that it dynamically guides the expansion of F^1 toward error states of the precomponents of F^0. This can be done by using a specialized BDD operation that splits the precomponent of F^1 according to the Hamming distance to the error states. The complexity of this operation, however, is exponential in the size of the BDD representing the error states and the size of the BDD representing the precomponent of F^0. Due to the dynamic programming used by the BDD package, the average complexity may be much lower. However, this does not seem to be the case in practice.

The third strategy is the one chosen for 1-GFTP. It expands F^1 blindly, but then prunes SAs from the precomponent of F^1 not used to recover error states of F^0. Thus, it uses an indirect approach to guide the expansion of F^1. We expect this strategy to work well even if the *absolute position* of error states is non-local. However, the strategy assumes that the *relative position* of error states is local in the sense that the SAs in F^1 in expansion i of F^0 are relevant for recovering error states in expansion $i+1$ of F^0. In addition, we still have an essential problem to solve: to expand F^0 or F^1. There are two extremes.

1. *Expand F^1 until first recovery of f^0.* Compute a complete partitioned backward precomponent of F^0, expand F^1 until some partition in f^0 has recovered error states and add the partition with least h-value to F^0.

2. *Expand F^1 until best recovery of f^0.* Compute a complete partitioned backward precomponent of F^0, expand F^1 until the partition of f^0 with lowest h-value has recovered error states and add this partition to F^0. If none of these error states can be recovered then consider the partition with second lowest h-value and so on.

It turns out that neither of these extremes work well in practice. The first is too conservative. It may add a partition with a high h-value even though a partition with a low h-value can be recovered given just a few more expansions of F^1. The second strategy is too greedy. It ignores the complexity of expanding F^1 in order to recover error states of the partition of f^0 with lowest h-value. Instead, we consider a mixed strategy: spend half of the last expansion time on recovering error states of the partition of f^0 with lowest h-value and, in case this is impossible, spend one fourth of the last expansion time on recovering error states of the partition of f^0 with second lowest h-value, and so on.

The 1-GFTP algorithm is shown in Figure 6.4. The keys in maps are sorted ascendingly. The instantiation of F^0 and F^1 of 1-GFTP is similar to 1-FTP except that the states in C^0 are partitioned with respect to their associated h-value. Initially the map entry, $\mathbf{C}^0[h_{goal}]$ is assigned to the goal states. [1] The variable t stores the duration of the previous expansion. Initially, it is given a small value ϵ. In each iteration of the main loop (l. 4-22), the precomponents f^0 and f^1 are computed and added to F^0 and F^1. First, the start time t_s is logged by reading the current time t_{CPU} (l. 5). Then a map $\mathbf{PC}$ holding a complete partitioned precomponent candidate of F^0 is computed by PRECOMPFTP (l.

[1] To simplify the presentation, we assume that all goal states have identical h-value. A generalization of the algorithm is trivial.

function 1-GFTP(s_0, G)
1 $F^0 \leftarrow \emptyset;\ \mathbf{C}^0[h_g] \leftarrow G$
2 $F^1 \leftarrow \emptyset;\ C^1 \leftarrow G$
3 $t \leftarrow \epsilon$
4 **while** $s_0 \notin C^0$
5 $\quad t_s \leftarrow t_{CPU}$
6 $\quad \mathbf{PC} \leftarrow$ PRECOMPFTP$(\mathbf{C}^0)$
7 $\quad f^0 \leftarrow \emptyset;\ f_c^0 \leftarrow \emptyset$
8 $\quad \mathbf{f}_c^1 \leftarrow emptyMap$
9 $\quad i \leftarrow 0$
10 $\quad$ **while** $f^0 = \emptyset \wedge i < |\mathbf{PC}|$
11 $\qquad i \leftarrow i + 1;\ t \leftarrow t/2$
12 $\qquad f_c^0 \leftarrow f_c^0 \cup \mathbf{PC}[i]$
13 $\qquad \langle \mathbf{f}_c^1, f^0 \rangle \leftarrow$ EXPANDTIMED$(f_c^0, \mathbf{f}_c^1, C^1, t)$
14 $\quad$ **if** $f^0 = \emptyset$ **then**
15 $\qquad \langle \mathbf{f}_c^1, f^0 \rangle \leftarrow$ EXPANDTIMED$(f_c^0, \mathbf{f}_c^1, C^1, \infty)$
16 $\quad t \leftarrow t_{CPU} - t_s$
17 $\quad$ **if** $f^0 = \emptyset$ **then return** "no solution exists"
18 $\quad f^1 \leftarrow$ PRUNEUNUSED$(\mathbf{f}_c^1, f^0)$
19 $\quad F^1 \leftarrow F^1 \cup f^1;\ C^1 \leftarrow C^1 \cup$ STATES(f^1)
20 $\quad F^0 \leftarrow F^0 \cup f^0$
21 $\quad$ **for** $j = 1$ **to** i
22 $\qquad \mathbf{C}^0[h_j] \leftarrow \mathbf{C}^0[h_j] \cup$ STATES$(f^0 \cap \mathbf{PC}[h_j])$
23 **return** $\langle F^0, F^1 \rangle$

Figure 6.4: The 1-GFTP algorithm.

6). For each entry in $\mathbf{C}^0$, PRECOMPFTP inserts the preimage in $\mathbf{PC}$ of each partition of a disjunctive branching partitioning of the transition relation of primary effects. We assume that this partitioning has m subrelations $R_1, \cdots R_m$ where the transitions represented by R_i are associated with a change δh_i of the h-value (in forward direction). The inner loop (l. 10-13) of 1-GFTP expands the two candidates f_c^0 and f_c^1 for f^0 and f^1. In each iteration, a partition of the partitioned precomponent $\mathbf{PC}$ is added to f_c^0 (l. 12).[2] The function EXPANDTIMED expands f_c^1. In iteration i, the time out bound of the expansion is $t/2^i$. EXPANDTIMED returns early if

[2]Recall that $\mathbf{PC}$ is traversed ascendingly such that the partition with lowest h-value is added first.

1. a precomponent f^0 in the candidate f_c^0 is found where all error states are recovered (l. 5 and l. 11), or

2. f_c^1 has reached a fixed point.

The preimage added to f_c^1 in iteration i of EXPANDTIMED is stored in the map entry $\mathbf{f}_c^1[i]$ in order to prune SAs not used for recovery.

function PRECOMPFTP$(\mathbf{C}^0)$
$\quad\mathbf{PC} \leftarrow \mathit{emptyMap}$
$\quad$**for** $i = 1$ **to** $|\mathbf{C}^0|$
$\quad\quad$**for** $j = 1$ **to** m
$\quad\quad\quad SA \leftarrow \text{PREIMGSA}_j(\mathbf{C}^0[h_i]) \setminus C^0 \times Act$
$\quad\quad\quad \mathbf{PC}[h_i - \delta h_j] \leftarrow \mathbf{PC}[h_i - \delta h_j] \cup SA$
$\quad$**return** $\mathbf{PC}$

Eventually f_c^0 may contain all the SAs in **PC** without any of these being recoverable. In this case 1-GFTP expands f_c^1 (l. 15) untimed.

function EXPANDTIMED$(f_c^0, \mathbf{f}_c^1, C^1, t)$
1 $t_s \leftarrow t_{CPU}$
2 $\mathit{Oldf}_c^1 \leftarrow \bot$
3 $i \leftarrow |\mathbf{f}_c^1|$
4 $\mathit{recovS} \leftarrow \text{STATES}(f_c^1) \cup C^1$
5 $f^0 \leftarrow f_c^0 \setminus \text{PREIMGSA}_f(\overline{\mathit{recovS}})$
6 **while** $f^0 = \emptyset \wedge \mathit{Oldf}_c^1 \neq f_c^1 \wedge t_{CPU} - t_s < t$
7 $\quad\mathit{Oldf}_c^1 \leftarrow f_c^1$
8 $\quad i \leftarrow i + 1$
9 $\quad\mathbf{f}_c^1[i] \leftarrow \text{PREIMGSA}(\mathit{recovS}) \setminus \mathit{recovS} \times Act$
10 $\quad\mathit{recovS} \leftarrow \text{STATES}(f_c^1) \cup C^1$
11 $\quad f^0 \leftarrow f_c^0 \setminus \text{PREIMG}_f(\overline{\mathit{recovS}})$
12 **return** $\langle \mathbf{f}_c^1, f^0 \rangle$

If f_c^1 has reached a fixed point but no recoverable precomponent f^0 exists, no 1-fault tolerant plan exists and 1-GFTP returns with "no solution exists" (l. 17). Otherwise, f_c^1 is pruned for SAs of states not used to recover the SAs in f^0 (l. 18). This pruning is computed by PRUNEUNUSED that traverses backward through the preimages of $\mathbf{f}_c^1$ and marks states that either are error states of SAs in f^0, or states needed to recover previously marked states.

function PRUNEUNUSED$(\mathbf{f}_c^1, f^0)$
$err \leftarrow \text{SAIMG}_f(f^0)$
$img \leftarrow \emptyset;\ marked \leftarrow \emptyset$
for $i = |\mathbf{f}_c^1|$ **to** 1
  $\mathbf{f}_c^1[i] \leftarrow \mathbf{f}_c^1[i] \cap ((err \cup img) \times Act)$
  $marked \leftarrow marked \cup \text{STATES}(\mathbf{f}_c^1[i])$
  $img \leftarrow \text{SAIMG}(\mathbf{f}_c^1[i])$
return $f_c^1 \cap (marked \times Act)$

The function $\text{SAIMG}(\pi)$ and $\text{SAIMG}_f(\pi)$ computes the image states of a set of SAs π for primary and secondary effects respectively.

$$\text{SAIMG}(\pi) \equiv \{s' : \exists \langle s, a\rangle \in \pi \,.\, s \xrightarrow{a} s'\} \tag{6.1}$$

$$\text{SAIMG}_f(\pi) \equiv \{s' : \exists \langle s, a\rangle \in \pi \,.\, s \overset{a}{\rightsquigarrow} s'\} \tag{6.2}$$

The updating of F^0 and F^1 of 1-GFTP (l. 19-22) is similar to 1-FTP, except that $\mathbf{C}^0$ is updated by iterating over $\mathbf{PC}$ and picking SAs in f^0. Notice that in this iteration h_j refers to the keys of $\mathbf{PC}$. We claim without proof that 1-GFTP is *sound*, *complete*, and *terminating*.

The specialized algorithms can be generalized to n faults by adding more recovery plans $F^n, F^{n-1}, \cdots, F^0$. For n-GFTP all of these recovery plans would be indirectly guided by the expansion of F^n. The algorithm is illustrated in Figure 6.5

6.3 Experimental Evaluation

The purpose of the experimental evaluation is not only to compare the performance of the developed algorithms, but also to investigate the properties of fault tolerant planning in significant real-world domains. The algorithms 1-FTP, 1-GFTP, 1-FTP$_S$, and 1-GFTP$_S$ have been implemented in the BIFROST 0.7 search engine. All experiments have been carried out in the experimental setting described in Appendix A. As usual, we represent the parameter setting of the BuDDy package by the number of allocated BDD nodes in the unique table (n) and the number of allocated BDD nodes in the operator caches (c). Time is measured in seconds and the size of a BDDs is measured in number of BDD nodes.

6.3.1 Unguided Search

We first focus on unguided search and study four fault tolerant planning domains. Two of these, DS1 and PSR, are models of real-world domains.

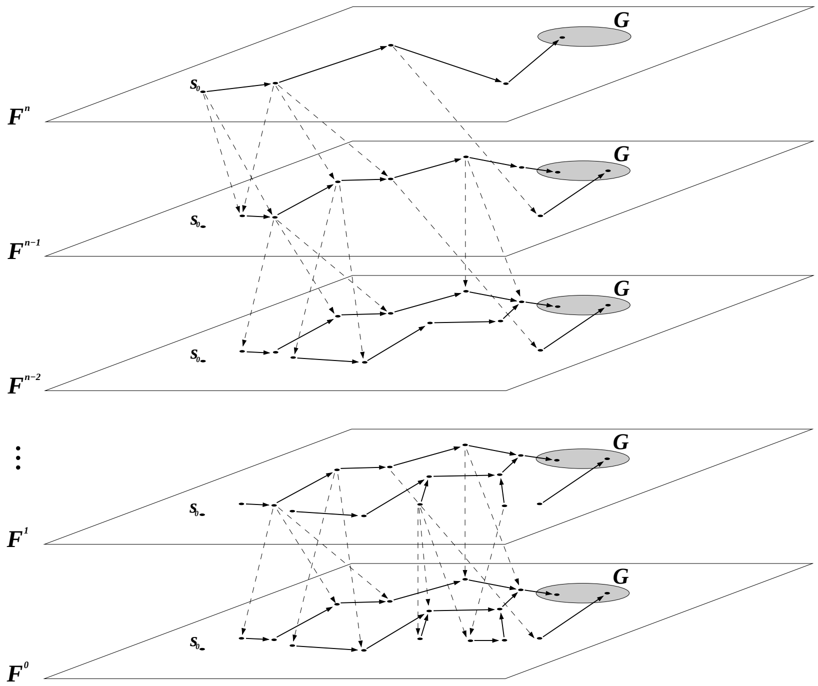


Figure 6.5: An example of $F^n, \cdots, F^0$ produced by a specialized n-fault tolerant planning algorithm. Primary and secondary effects of actions are drawn with solid and dashed lines, respectively.

DS1

DS1 is based on an SMV encoding [131] of the Livingstone model [173] used by the Remote Agent for NASA's Deep Space One probe. The Livingstone model describes the electrical system of the spacecraft. It consists of a system bus and a number of units connected to the bus. These units include a power distribution subsystem, a Ion Propulsion System (IPS), Propulsion Drive Electronics (PDE), a Reaction Control System (RCS), Attitude Control System (ACS), Star Tracker Unit (SRU), and a MICAS camera. We recast the SMV encoding as a fault tolerant planning problem in NADL$^+$. Each bus-command is an action. The primary effect of the command is the changes it causes on the electrical system given that all units work correct. The secondary effect of an action is one of the two faults F2 and F4 considered in the Remote Agent Experiment [124].

F2 : camera or pasm switch is recoverably stuck on/off.

F4 : an x-z thruster valve is permanently stuck closed.

In addition to these two faults, the Remote Agent Experiment considered two other errors. We are not modeling these since no 1-fault tolerant plan exists when taking all four faults into account. The following simplifications have been made in the NADL$^+$ model of the SMV description

1. we assume that the state of components is known,
2. attitude errors are assumed to be deterministically computable,
3. relative thrust is assumed to be low or nominal if a valve is stuck otherwise nominal,
4. redundant state variables in the SMV model have been removed. [3]

The NADL$^+$ encoding of the domain has 84 Boolean state variables. We consider generating a 1-fault tolerant plan from an initial state where the IPS is in standby mode, the MICAS camera is "off", and the pasm switch is "on". The goal is to reach a state where the IPS is in thrusting mode, the MICAS camera is "on", and the pasm switch is "off". The BDD package parameters are n = 1M and c = 100K. The threshold for merging partitions of a disjunctive transition relation partitioning is 5000. The total size of the transition relation is 104881 and is computed in 0.42 seconds. The size of the solution is 535 and the total CPU time is 1.15 seconds. The experiment shows that a BDD encoding is very efficient for the kind of constraints modeled by DS1. Despite a fairly large and dense model, a disjunctive transition relation is fast to compute. In addition, a 1-fault tolerant plan for a non-trivial problem in this domain is small and can be generated in less than a second. The experiment demonstrates that BDD-based fault tolerant planning is mature to be applied on significant real-world problems. An important lesson to learn from the investigation of DS1 is that even 1-fault tolerance imposes a strong restriction on a physical system. No 1-fault tolerant plan exists for the problem if all of the original four failures are considered.

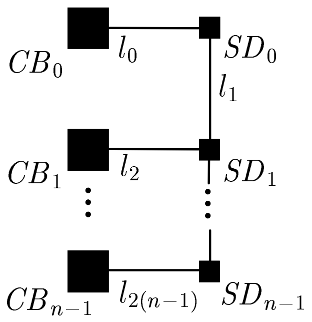


Figure 6.6: The linear PSR domain.

[3]An automatic approach for doing this has been developed in [178].

PSR

The PSR domain is described in Section 5.3.1. The primary effect of the Open and Close actions on switches is that the switches open and close accordingly. The secondary effect is that they break and get stuck in their current position. We compare the performance of 1-FTP and 1-FTP$_S$ in two versions of the domain. The first, is the "simple" domain described Section 5.3.1. In the initial state, all switches are open and the goal is to feed all lines. 1-FTP and 1-FTP$_S$ solve this problem in 6.8 and 11.25 seconds, respectively (0.98 seconds is used on memory allocation, n = 1M and c = 700K).

The second version of the domain is the linear network shown in Figure 6.6. Again, the initial state is that all switches are open, and the goal is to feed all lines. The result are shown in Figure 6.7. The BDD package parameters are $n = 15M$ and $c = 500K$ and 3.38

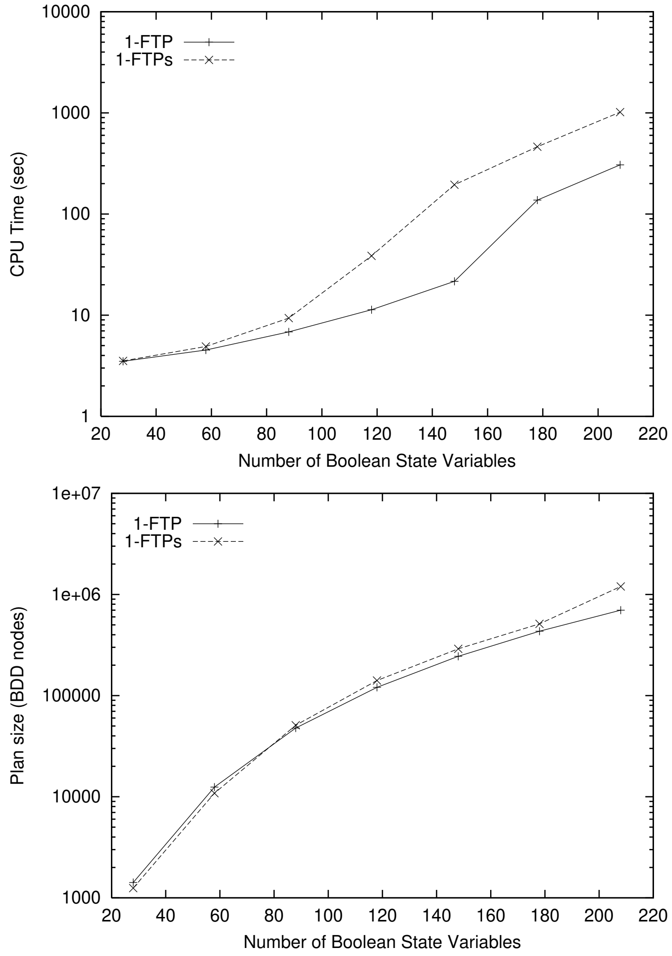


Figure 6.7: Results of the PSR problems.

seconds are used on memory allocation. 1-FTP performs significantly better than 1-FTP$_S$ on this problem. Interestingly, the performance difference is not reflected by the plan sizes. However, this may be an artifact caused by the fact that the plan size for 1-FTP is a sum of the size of two BDDs, while the plan size for 1-FTP$_S$ is the size of a single BDD. Similarly to the DS1 domain, 1-fault tolerance imposes a strong constraint on the PSR domain. For most configurations, where a few units already have failed, no 1-fault tolerant plan exists.

Power Plant

The power plant domain is shown in Figure 6.8 and originates in [93]. The task is to execute

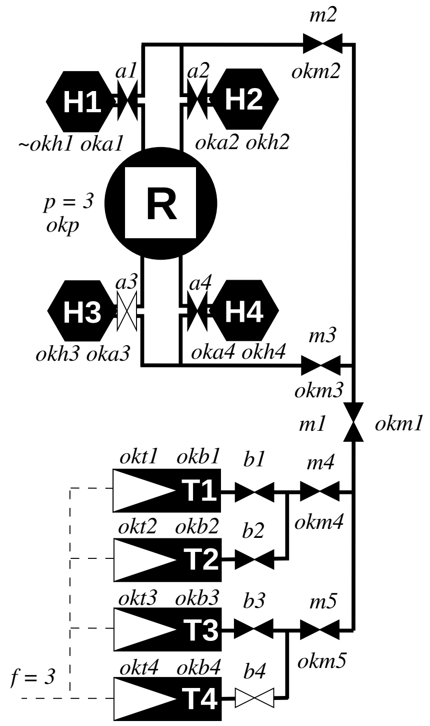


Figure 6.8: The power plant domain. An open valve is drawn solid and allows water or steam to flow through it. In the depicted state, a failure of heat exchanger 1 is assumed just to have happened.

the correct control actions in order to bring the plant from some bad state, where the plant is unsafe or not working properly, to some good state, where the plant satisfies its safety and activity requirements. A single reactor R is surrounded by four heat exchangers H1, H2, H3 and H4. The heat exchangers produce high pressure steam to the four electricity generating turbines T1, T2, T3 and T4. The heat exchangers can fail and leak radioactive substances from the internal water loop to the external steam loop. If this happens, the

	1-FTP		1-FTP$_S$	
$size$	t_{total}	$\|sol\|$	t_{total}	$\|sol\|$
40	6.1	65K	8.7	62K
80	157.8	1.2M	189.4	1.5M

Figure 6.9: Results of the power plant experiment. The total CPU time and plan size is given by t_{total} and $|sol|$, respectively. The size of the problem is the number of Boolean state variables.

blocking valve ($a1$, $a2$, $a3$ or $a4$) of the heat exchanger must be closed. However, these valves can fail too, in which case the valves $m2$, $m3$ or $m1$ are used. Similarly, if turbines fail, they must be shut down by closing one of the valves $b1$, $b2$, $b3$ or $b4$, or $m4$, $m5$ and $m1$. The energy production p of the plant can either be 0,1,2,3 or 4 units of energy per time unit. The production must be adjusted to fit the demand f, if possible. A heat exchanger can only transfer enough energy to a single turbine, and a single turbine can only produce one unit of energy per time unit. The initial state is shown in Figure 6.8. A failure of heat exchanger 1 is a assumed to have just happened.

We compare the performance of 1-FTP and 1-FTP$_S$ in two versions of the domain. The first considers controlling a single power plant. The second considers controlling two power plants simultaneously. The results are shown in Figure 6.9. In both experiments, the parameters of the BDD package are $n = 15M$ and $c = 500K$. The time spent on memory allocation is 3.4 seconds. 1-FTP has a slightly better performance than 1-FTP$_S$. However, both algorithms suffer from a large growth rate of the BDDs representing the frontier of the backward search. Again, 1-fault tolerant plans turns out to be hard to generate. Even though the system is highly redundant, 1-fault tolerant plans only exist for simple malfunctions like the one investigated in this experiment.

Beam Walk

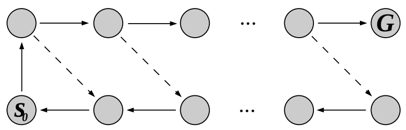


Figure 6.10: The Beam Walk domain. Solid edges denote primary effects of the move action, while dashed edges denote secondary effects.

The Beam Walk domain was introduced in [36] and considers a robot walking on a beam. The primary effect of the move action is that the robot moves one step forward on the beam. The secondary effect is that it falls down from the beam. The domain is shown in Figure 6.10. The BeamWalk domain represents a worst case scenario for 1-FTP and 1-FTP$_S$ since a fault in the last step to reach the goal causes a transition to the state furthest away from the goal. Both algorithms must iterate over all states before a solution is found. The results are shown in Figure 6.11. As expected, both algorithms have a limited

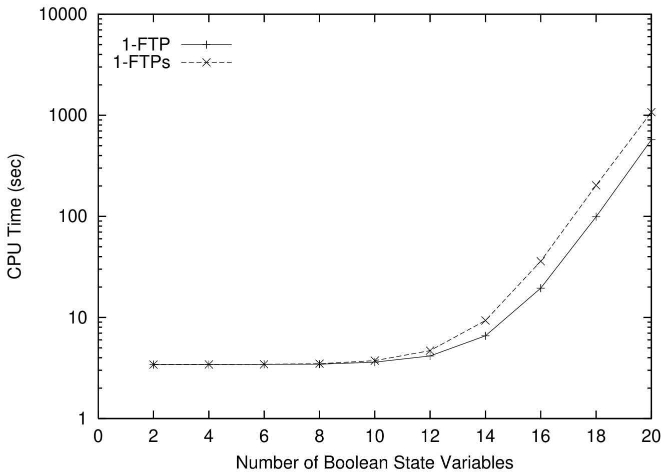


Figure 6.11: Results of the BeamWalk experiments.

performance in this domain. Again, however, we observe a slightly better performance of 1-FTP.

6.3.2 Guided Search

The main purpose of the experiments in this section is to study the difference between 1-GFTP and 1-GFTP$_S$. In particular, we are interested in investigating how sensitive these algorithms are to non-local error states and to what extent we may expect this to be a problem in practice. We study 3 domains, of which SIDMAR descends from a real-world study.

LV

The LV domain is an artificial domain and has been designed to demonstrate the different properties of 1-GFTP and 1-GFTP$_S$. It is an $m \times m$ grid world with initial state $(0, m-1)$ and goal state $(\lfloor m/2 \rfloor, \lfloor m/2 \rfloor)$. The actions are Up, Down, Left, and Right. Above the $y = x$ line, actions may fail causing the x and y position to be swapped. Thus, error states are mirrored in the $y = x$ line. A 9×9 instance of the problem is shown in Figure 6.12.

The essential property is that error states are non-local, but that two states close to each

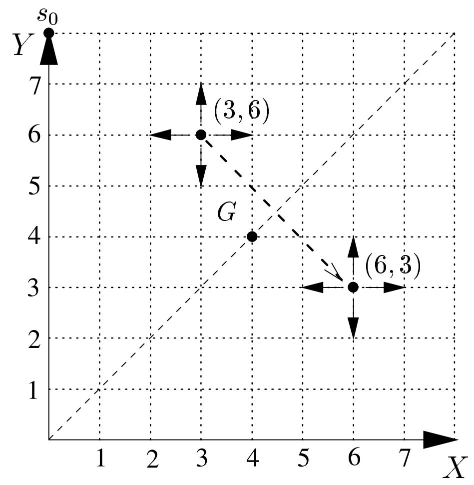


Figure 6.12: The 9×9 instance of the LV domain.

other also have error states close to each other. This is the assumption made by 1-GFTP, but not 1-GFTP$_S$ that requires error states to be local. The heuristic value of a state is the Manhattan distance to the initial state. The BDD package parameters are $n = 5M$ and $c = 500K$. Memory allocation takes 1.4 seconds. The results are shown in Figure 6.13. As

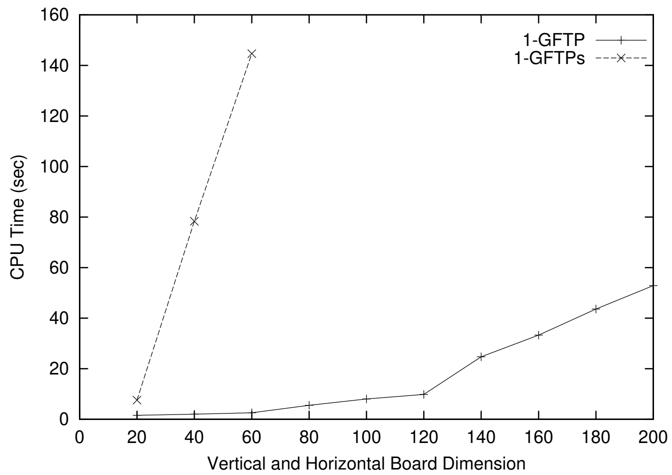


Figure 6.13: Results of the LV experiments.

depicted, the performance of 1-GFTP$_S$ degrades very fast with m due to the misguidance of the heuristic for the recovery part of the plan. Its total CPU time is more than 500 seconds after the first three experiments. 1-GFTP$_S$ is fairly unaffected by the error states.

To explain this, consider how the backward search proceeds from the goal state. The guided precomponents of F^0 will cause this plan to beam out toward the initial state. Due to the relative locality of error states, the pruning of F^1 will cause F^1 to beam out in the opposite direction. Thus, both F^0 and F^1 remain small during the search.

8-Puzzle

The 8-Puzzle further demonstrates this difference between 1-GFTP and 1-GFTP$_S$. We consider a non-deterministic version of the 8-Puzzle where the secondary effects are self loops. Thus, error states are the most local possible. We use the usual sum of Manhattan distances of tiles as an heuristic for the distance to the initial state. The experiment compares the performance of 1-FTP, 1-GFTP, 1-FTP$_S$, and 1-GFTP$_S$. The BDD package parameters are $n = 1M$ and $c = 100K$. Memory allocation takes 0.29 seconds. The number of Boolean state variables is 35 in all experiments. The results are shown in Figure 6.14. Again, 1-FTP performs substantially better than 1-FTP$_S$. The guided algorithms 1-GFTP and 1-GFTP$_S$ have much better performance than the unguided algorithms. Due to local error states, however, there is no substantial performance difference between these two algorithms. As depicted, 1-FTP is slightly faster than 1-GFTP$_S$ in the experiment with a minimum deterministic solution length of 14. For such small problems, we may expect to see this since 1-FTP only expands the recovery plan when needed while 1-GFTP$_S$ expands the recovery part of its plan in each iteration.

SIDMAR

The final experiments are on the SIDMAR domain introduced in Section 5.3.1. The purpose of these experiments is to study the robustness of 1-GFTP and 1-GFTP$_S$ to the kind of errors found in real-world domains. The primary effects of actions are to move, lift and perform treatments of ladles on machines. The secondary effects are that machines break permanently and moves fail. We consider casting two ladles of steel. The heuristic is the sum of machine treatments carried out on the ladles. The experiment compares the performance of 1-FTP, 1-GFTP, 1-FTP$_S$, and 1-GFTP$_S$. The BDD package parameters are $n = 5M$ and $c = 500K$. Memory allocation takes 1.41 seconds. The number of Boolean state variables is 47 in all experiments. The results are shown in Figure 6.15. Missing data points indicates that the associated algorithm spent more than 500 seconds trying to solve the problem. The only algorithm with good performance is 1-GFTP. The experiment indicates that real-world domains may have non-local error states that limits the performance of 1-GFTP$_s$. Also notice that this is the only domain where 1-FTP does not outperform 1-FTP$_S$. In this domain, 1-FTP seems to be finding complex plans that fulfills

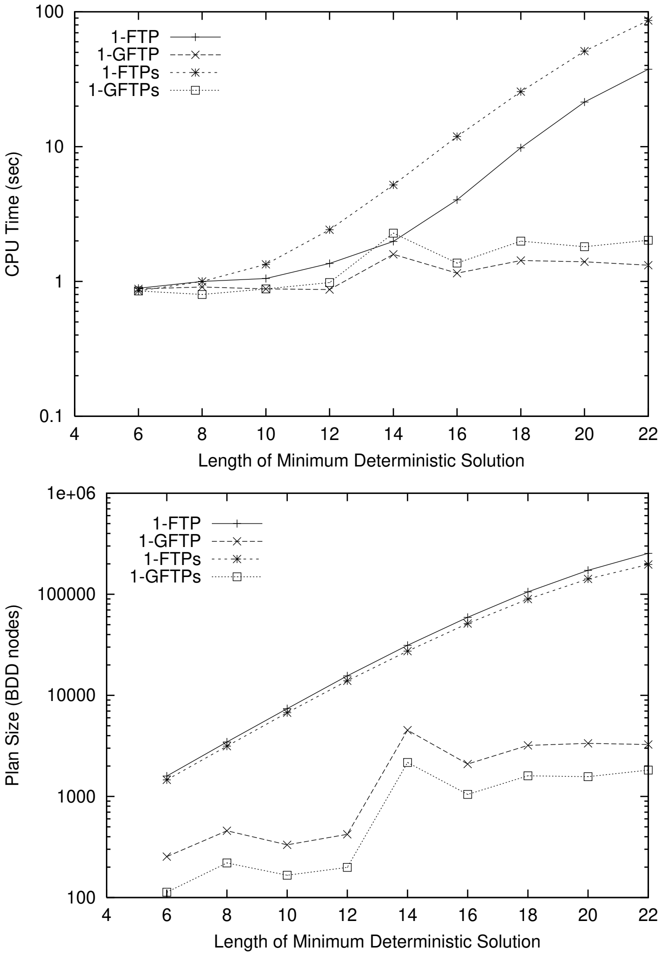


Figure 6.14: Results of the 8-Puzzle experiments.

that the recovery plan is minimum. Thus, the strategy of 1-FTP to keep the recovery plan as small as possible does not seem to be an advantage in general.

6.4 Conclusion

The experimental evaluation shows that 1-GFTP consistently outperforms its strong algorithm counter part 1-GFTP$_S$ and in particular is robust to non-local error states. Our investigation of real-world domains suggests that such error states exist and are caused by permanent failures. Despite the blind search of 1-FTP, it often outperforms its strong algorithm counter part 1-FTP$_S$ since it may avoid producing large recovery plans.

The experimental evaluation of DS1, Power Plant, and PSR further shows that 1-fault

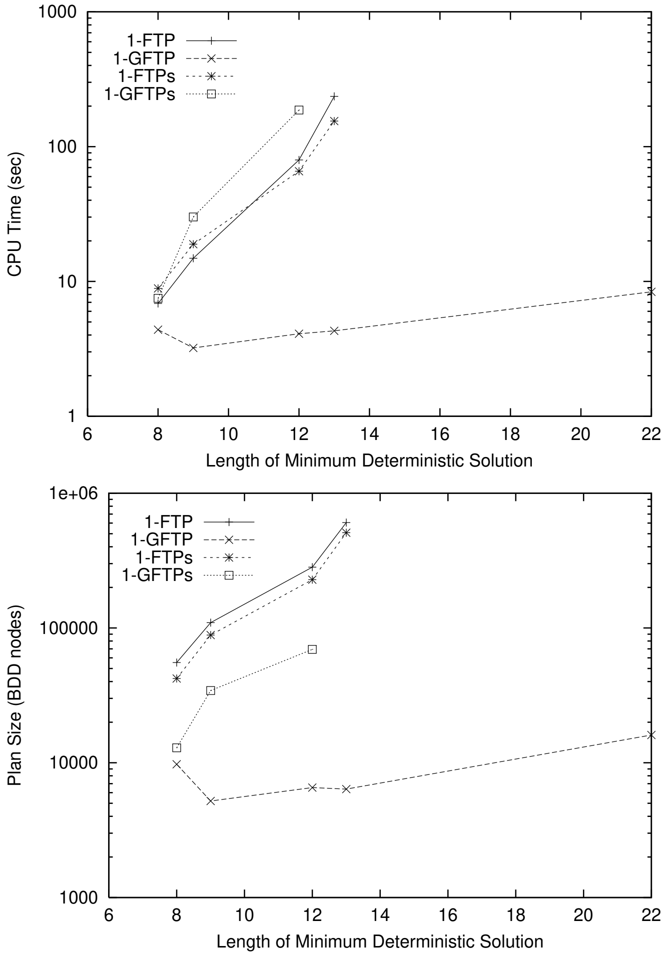


Figure 6.15: Results of the SIDMAR experiments.

tolerant plans often do not exist even for highly redundant physical systems. This suggests that a fruitful direction for future work is to define classes of fault tolerant plans that are more relaxed than 1-fault tolerant plans. Another direction of work is to consider fault tolerant plans that are adjusted to the likelihood of faults. The more likely a fault is, the more robust the fault tolerant plan should be for it. Finally, it seems fairly simple to allow non-deterministic primary effects. In this case the strong precomponent would be natural to use to expand the nonfaulting and recovery part of the plan.

6.5 Summary

In this chapter, we have introduced a new class of non-deterministic plans called fault tolerant plans. Fault tolerant plans address domains where non-determinism is caused by infrequent errors. For such domains strong and strong cyclic solutions seldom exist since any action may fail. Fault tolerant plans relax this problem by being robust only to a limited number of faults occurring during execution. Fault tolerant plans can be synthesized with the strong algorithm by reducing a fault tolerant planning problem to a non-deterministic planning problem. However, due to non-local error states, we introduce a specialized guided algorithm called 1-GFTP that decouples the guiding of the fault tolerant and recovery part of the plan. The experimental evaluation indicates that this decoupling may be very helpful for obtaining good performance in real-world domains.

Chapter 7

Adversarial Planning

In the previous chapter, we identified faults as a major source of non-determinism in real-world domains. In this chapter, we introduce a new framework called *adversarial planning* [94] to address domains where non-determinism is caused by simultaneous actions of a controllable system agent and an uncontrollable and possibly hostile environment agent. Each state is associated with a set of actions that are applicable by the system agent and a set of actions that are applicable by the environment agent. In each execution step, the two agents select one of their applicable actions. They have no knowledge about the action selected by the other agent. The two actions form a *joint action* that causes a transition to a new state.

We begin our description of adversarial planning in Section 7.1 by modifying the non-deterministic domain model introduced in Section 3.2 to represent system and environment actions. We then demonstrate that for these domains there exist plans that are more powerful than weak and strong cyclic plans. These adversarial plans can be generated by reasoning explicitly about environmental actions. In Section 7.2, we introduce two new algorithms for synthesizing *weak adversarial plans* and *strong cyclic adversarial plans*. We prove that, in contrast to strong cyclic plans, strong cyclic adversarial plans guarantee goal achievement independent of the environment behavior if actions are selected randomly from the plan. Similarly, we prove that given actions are selected randomly from the plan, weak adversarial plans improve the quality of weak plans by guaranteeing that there is a non-zero probability of reaching a goal state independent of the behavior of the environment. In Section 7.4, the algorithms are evaluated experimentally both in terms of their computational efficiency and in terms of the quality of the produced plans. Finally, we draw conclusions in Section 7.5.

7.1 Adversarial Planning Problems

An adversarial planning domain has two active agents: a *system* and an *environment*. The task is to construct plans for the system in order for it to achieve a goal. The environment may be an intelligent adversary (or it may simply be an advantage to assume that) who is fully informed about the structure of the domain and the limitations of the system's ability to construct plans.[1]

An adversarial planning domain is a non-deterministic planning domain with a set of controllable system actions and a set of uncontrollable environment actions. System and environment actions are synchronous. The transition relation of the domain describes the effects of joint system and environment actions. The transition relation is deterministic to reflect that the only source of non-determinism is uncontrollable environment actions.

Definition 7.1 (Adversarial Planning Domain) *An adversarial planning domain is a tuple $\langle S, Act_s, Act_e, \rightarrow\rangle$ where S is a finite set of states, Act_s is a finite set of system actions, , Act_e is a finite set of environment actions, and $\rightarrow \subseteq S \times Act_s \times Act_e \times S$ is a deterministic transition relation of joint system and environment actions. Instead of $(s, a_s, a_e, s') \in \rightarrow$, we write $s \xrightarrow{a_s,a_e} s'$.*

Adversarial planning domains can be described in NADL$^+$. The set of applicable actions of a state s are defined by the functions

$$\begin{aligned} \text{A}\textsc{pp}(s) &\equiv \{\langle a_s, a_e\rangle : \exists s' . s \xrightarrow{a_s,a_e} s'\} && (7.1)\\ \text{A}\textsc{pp}_s(s) &\equiv \{a_s : \exists a_e . \langle a_s, a_e\rangle \in \text{A}\textsc{pp}(s)\} && (7.2)\\ \text{A}\textsc{pp}_e(s) &\equiv \{a_e : \exists a_s . \langle a_s, a_e\rangle \in \text{A}\textsc{pp}(s)\} && (7.3) \end{aligned}$$

where $\text{A}\textsc{pp}(s)$, $\text{A}\textsc{pp}_s(s)$, and $\text{A}\textsc{pp}_e(s)$ give the set of joint-actions, system actions, and environments actions applicable in s, respectively. It is required that system and environment actions are independent at each state. Otherwise the system can indirectly control the environment by making some of its action unapplicable and vice versa. Thus

$$\text{A}\textsc{pp}(s) = \text{A}\textsc{pp}_s(s) \times \text{A}\textsc{pp}_e(s). \quad (7.4)$$

The set of states that can be reached from s by some joint action from s involving the system action a_s is given by

$$\text{N}\textsc{ext}_s(s, a_s) \equiv \{s' : \exists a_e . s \xrightarrow{a_s,a_e} s'\}. \quad (7.5)$$

An adversarial planning problem is defined by an initial state and a set of goal states of the system.

[1] This is a standard assumption in e.g. matrix games and extensive form games [127].

Definition 7.2 (Adversarial Planning Problem) *An adversarial planning problem is a tuple $\langle \mathcal{D}, s_0, G \rangle$ where $\mathcal{D}$ is an adversarial planning domain, $s_0 \in S$ is an initial states, and $G \subseteq S$ is a set of goal states.*

An *adversarial plan* is a plan for the system represented in the usual way as a set of state-action pairs.

Definition 7.3 (System State-Action Pair (SSA)) *Let $\mathcal{D}$ be an adversarial planning domain. A system state-action pair $\langle s, a_s \rangle$ of $\mathcal{D}$ is a state s of $\mathcal{D}$ associated with an applicable system action $a_s \in \text{APP}_s(s)$*

Definition 7.4 (System Plan) *Let $\mathcal{D}$ be an adversarial planning domain. A system plan π_s for $\mathcal{D}$ is set of SSAs of $\mathcal{D}$.*

We will use π_s to denote system plans and often refer to them as *adversarial plans*.

Example 7.1 For the the adversarial planning problem shown in Figure 7.1(a), we have

$$\begin{aligned}
S &= \{I, F, D, U, G\}, \\
s_0 &= I, \\
G &= \{G\}, \\
Act_s &= \{+s, -s\}, \\
Act_e &= \{+e, -e\}, \\
\rightarrow &= \{\langle I, +s, -e, F\rangle, \langle I, -s, -e, U\rangle, \langle F, -s, -e, F\rangle, \langle F, +s, +e, F\rangle, \\
&\quad \langle F, +s, -e, G\rangle, \langle F, -s, +e, G\rangle, \langle U, -s, -e, D\rangle, \langle U, +s, +e, U\rangle, \\
&\quad \langle U, +s, -e, G\rangle, \langle U, -s, +e, G\rangle\}.
\end{aligned}$$

Notice that this transition relation fulfills the requirement $\text{APP}(s) = \text{APP}_s(s) \times \text{APP}_e(s)$ for any state s. The state D is a dead end, since the goal is unreachable from D. This introduces an important difference between F and U that captures a main aspect of the adversarial planning problem. We can view the two states F and U as states in which the system and environment have different opportunities. Observe that the system "wins", i.e., reaches the goal, only if the sign of the two actions in the joint action are different. Otherwise it "loses" since there is no transition to the goal with a joint action where the actions have the same sign. The goal is reachable from both F and U. However, the consequences of losing is different for F and U. In F, losing causes a transition back to F. Thus, the goal is still reachable. In U, however, losing may cause a transition to the dead end D which makes it impossible to reach the goal in subsequent steps. Consider how an adversarial and

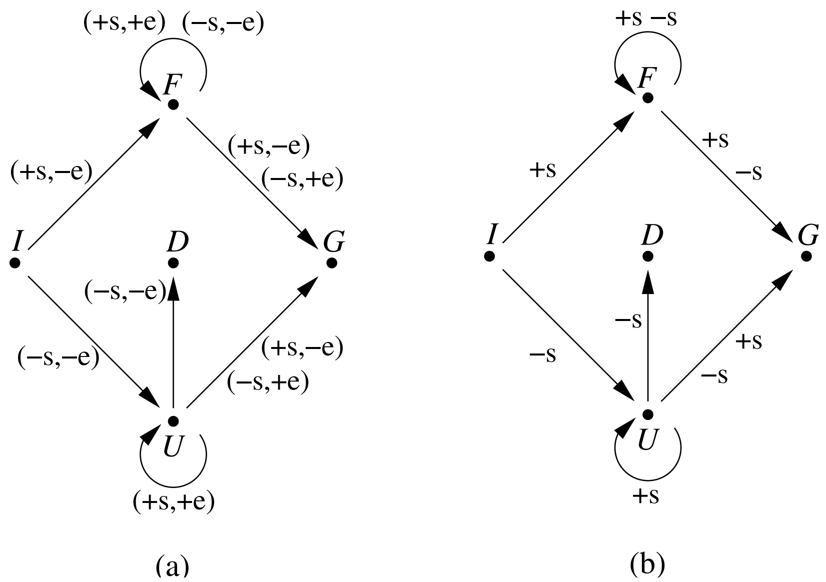


Figure 7.1: (a) An adversarial planning problem with five states $\{I, F, D, U, G\}$, an initial state I, and a single goal state G. The system and environment have actions $\{+s, -s\}$ and $\{+e, -e\}$, respectively. (b) The induced non-deterministic planning problem of the adversarial planning problem shown in (a) where the information about environment actions has been abstracted.

informed environment can take advantage of the possibility of reaching a dead end from U. Since this may happen if the system applies $-s$ in U, it is reasonable for the environment to assume that the system will always execute $+s$ in U. But now the environment can prevent the system from ever reaching the goal by always choosing action $+e$, so the system should completely avoid the state U. This example domain is important because it illustrates how an adversarial environment can act purposely to obstruct achievement of the goal. $\Diamond$

In order to introduce an execution model, we also need to define environment plans. These are sets of state-action pairs, where the action is an environment action.

Definition 7.5 (Environment State-Action Pair (ESA)) *Let $\mathcal{D}$ be an adversarial planning domain. An environment state-action pair $\langle s, a_e \rangle$ of $\mathcal{D}$ is a state s of $\mathcal{D}$ associated with an applicable environment action $a_e \in$ $\text{APP}_e(s)$*

Definition 7.6 (Environment Plan) *Let $\mathcal{D}$ be an adversarial planning domain. An environment plan π_e for $\mathcal{D}$ is set of ESAs of $\mathcal{D}$.*

We will use π_e to denote environment plans. The set of states covered by a system plan, an environment plan, and a combined plan is given by

$$\text{STATES}_s(\pi_s) \equiv \{s : \exists a_s . \langle s, a_s \rangle \in \pi_s\} \tag{7.6}$$

$$\begin{aligned} \textsc{States}_e(\pi_e) &\equiv \{s : \exists a_e \,.\, \langle s, a_e\rangle \in \pi_e\} && (7.7)\\ \textsc{States}(\pi_s, \pi_e) &\equiv \textsc{States}(\pi_s) \cap \textsc{States}(\pi_e). && (7.8)\end{aligned}$$

The set of actions of a plan associated with a state s is

$$\begin{aligned} \textsc{Act}_s(\pi_s, s) &\equiv \{a_s : \langle s, a_s\rangle \in \pi_s\} && (7.9)\\ \textsc{Act}_e(\pi_e, s) &\equiv \{a_e : \langle s, a_e\rangle \in \pi_e\}. && (7.10)\end{aligned}$$

The set of possible end states of a combined system plan π_s and an environment plan π_e is given by

$$\begin{aligned} \textsc{Closure}(\pi_s, \pi_e) = \{s' \notin \textsc{States}(\pi_s, \pi_e) : \exists s, a_s \in \textsc{Act}_s(\pi_s, s), \quad (7.11)\\ a_e \in \textsc{Act}_e(\pi_e, s) \,.\, s \xrightarrow{a_s, a_e} s'\}.\end{aligned}$$

We can now define the execution model of a system and environment plan.

Definition 7.7 (Exectution Model) *An execution model with respect to a system plan π_s and an environment plan π_e for the adversarial domain $\mathcal{D} = \langle S, Act_s, Act_e, \rightarrow\rangle$ is a Kripke structure $\mathcal{M}(\pi_s, \pi_e) = \langle S, R\rangle$ where*

- $S = \textsc{Closure}(\pi_s, \pi_e) \cup \textsc{States}(\pi_s, \pi_e) \cup G$,
- $\langle s, s'\rangle \in R$ *iff* $s \notin G$, $\exists a_s, a_e \,.\, \langle s, a_s\rangle \in \pi_s$, $\langle s, a_e\rangle \in \pi_e$, *and* $s \xrightarrow{a_s, a_e} s'$, *or* $s = s'$ *and* $s \in \textsc{Closure}(\pi_s, \pi_e) \cup G$.

The execution paths starting at s of the system plan π_s and environment plan π_e are given by

$$\textsc{Exec}(s, \pi_s, \pi_e) \equiv \{q : q \text{ is a path of } \mathcal{M}(\pi_s, \pi_e) \text{ and } q_0 = s\}. \quad (7.12)$$

An important question is if adversarial plans can be generated via a transformation to a non-deterministic planning problem and an application of an existing non-deterministic planning algorithm as were done with fault tolerant plans. One approach is to let the joint actions of the system and the environment form the actions of a corresponding non-deterministic planning problem. However, this would imply that joint actions are controllable which is inconsistent with the assumption that environment actions are uncontrollable. There does not seem to exist a simple solution to this problem except the obvious: to model the effect of environment actions as non-determinism of system actions. This transformation is defined as the induced non-deterministic planning problem of an adversarial planning problem.

Definition 7.8 (Induced Non-Deterministic Planning Problem) *Let $\mathcal{P} = \langle \mathcal{D}, s_0, G \rangle$ where $\mathcal{D} = \langle S, Act_s, Act_e, \rightarrow \rangle$ is an adversarial planning problem. The non-deterministic planning problem induced from $\mathcal{P}$ is $\mathcal{P}^{nd} = \langle \mathcal{D}^{nd}, s_0, G \rangle$ where $\mathcal{D}^{nd} = \langle S^{nd}, Act^{nd}, \rightarrow^{nd} \rangle$ and is given by*

- $S^{nd} = S$
- $Act^{nd} = Act_s$
- $s \xrightarrow{a_s}{}^{nd} s'$ *iff* $s \xrightarrow{a_s, a_e} s'$ *for some* $a_e \in \text{APP}_e(s)$.

Example 7.2 Figure 7.1(b) shows the induced non-deterministic planning problem of the adversarial planning problem described in Example 7.1. $\Diamond$

The *least restricted* environment plan is one where each state is associated with all applicable actions of the environment Let $\pi_e^\top$ denote the least restricted environment plan defined by

$$\pi_e^\top \equiv \{\langle s, a_e \rangle \, : \, a_e \in \text{APP}_e(s)\}. \tag{7.13}$$

For an environment plan to be *non-empty*, we require that it associates at least a single action with any state where the set of applicable actions is non-empty. Otherwise, it is trivial for the environment to construct a plan that for all executions prevent goal achievement. Let Π_e^+ denote the set of non-empty environment plans

$$\Pi_e^+ \equiv \{\pi_e \, : \forall s \, . \, \text{ACT}_e(\pi_e, s) \cap \text{APP}_e(s) \neq \emptyset\}. \tag{7.14}$$

A strong plan for the induced non-deterministic planning problem is an important class of adversarial plans. The fact that a strong solution exists means that the system is able to achieve its goal for any non-empty environment plan. If we regard the domain as a game between the system and environment, such plans are often referred to as *winning strategies* (e.g.,[3, 43]). Strong cyclic solutions to an induced non-deterministic planning problem, on the other hand, have limited value as shown in Example 7.3.

Example 7.3 There is no strong solution to the induced non-deterministic planning problem shown in Figure 7.1. The reason is that there does not exist a system action for F or U that guarantees a transition to G. A valid strong cyclic solution is

$$\pi_s = \{\langle I, +s \rangle, \langle I, -s \rangle, \langle F, +s \rangle, \langle F, -s \rangle, \langle U, +s \rangle\}.$$

This plan eventually reaches the goal if the environment is "friendly" and sometimes executes action $-e$ in state U. Such friendliness, however, is unlikely if the environment is an opponent. $\Diamond$

The problem with strong cyclic solutions is that they assume the environment uses the least restricted plan $\pi^\top$. This is also the case for weak plans.

Theorem 7.1 *Given an adversarial planning problem $\mathcal{P} = \langle \mathcal{D}, s_0, G\rangle$, a non-deterministic planning problem $\mathcal{P}^{nd} = \langle \mathcal{D}^{nd}, s_0^{nd}, G^{nd}\rangle$ induced from $\mathcal{P}$, and a plan π_s for $\mathcal{D}^{nd}$*

- *if π_s is a weak solution then $\mathcal{M}(\pi_s, \pi_e^\top), s_0 \models \texttt{EF}\, G$,*
- *if π_s is a strong cyclic solution then $\mathcal{M}(\pi_s, \pi_e^\top), s_0 \models \texttt{AGEF}\, G$,*
- *if π_s is a strong solution then $\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(\pi_s, \pi_e), s_0 \models \texttt{AF}\, G$.*

Proof. Follows directly from the definition of weak, strong cyclic, and strong plans and the definition of the induced non-deterministic planning problem. $\square$

As shown in Example 7.4, it turns out that there exists plans that are more powerful than weak and strong cyclic plans for adversarial planning problems.

Example 7.4 The strong cyclic plan

$$\pi_s = \{\langle I, +s\rangle, \langle I, -s\rangle, \langle F, +s\rangle, \langle F, -s\rangle, \langle U, +s\rangle\}$$

described in Example 7.3 can be improved by avoiding the state U. This is done by the following plan

$$\pi_s = \{\langle I, +s\rangle, \langle F, +s\rangle, \langle F, -s\rangle\}$$

which is guaranteed eventually to reach the goal for any non-empty strategy of the environment given that the system selects randomly between the actions in the plan. Or more precisely, there is a zero probability for any infinite plan not reaching a goal state. $\Diamond$

Adversarial planning introduces a class of adversarial weak and strong cyclic solutions that, similarly to strong solutions, are robust to any plan applied by the environment.

Definition 7.9 (Weak and Strong Cyclic Adversarial Plans) *Given an adversarial planning problem $\mathcal{P} = \langle \mathcal{D}, S_0, G\rangle$ and a plan π_s for $\mathcal{D}$*

- *π_s is a weak adversarial solution iff $\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(\pi_s, \pi_e), s_0 \models \texttt{EF}\, G$,*
- *π_s is a strong-cyclic adversarial solution iff $\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(\pi_s, \pi_e), s_0 \models \texttt{AGEF}\, G$.*

This can be done by generalizing the approach in Example 7.4 and prune states from weak and strong cyclic plans where the game between the system and the environment is unfair. We formalize this idea in the definition of a *fair state*. A state s is fair with respect to a set of states C and a plan π_s if s is not already a member of C and for each applicable environment action there exists a *counter action* in π_s such that the joint action leads into C.

Definition 7.10 (Fair State) *A state $s \notin C$ is fair with respect to a set of states C and a plan π_s iff* $\forall a_e \in \text{APP}_e(s) \,.\, \exists a_s \in \text{ACT}_s(\pi_s, s), s' \in C \,.\, s \xrightarrow{a_s, a_e} s'$.

For convenience, we define an *unfair* state to be a state that is not fair.

7.2 Adversarial Planning Algorithms

Weak and strong cyclic adversarial plans can be synthesized by modifying the ordinary weak and strong cyclic precomponents and employing the generic non-deterministic planning algorithm shown in Figure 3.8.[2]

The core computations of the precomponent functions are to find fair states of a plan π_s with respect to a set of states C and compute the preimage of system state-action pairs (SSAs) of a set of states C.

$$\begin{aligned} \text{FAIRSTATES}(\pi_s, C) &\equiv \{s \notin C \,:\, \forall a_e \in \text{APP}_e(s) \,.\, \exists a_s \in \text{ACT}_s(\pi_s, s), \quad (7.15)\\ &\qquad s' \in C \,.\, s \xrightarrow{a_s, a_e} s'\} \\ \text{PREIMGSSA}(C) &\equiv \{\langle s, a_s \rangle \,:\, \text{NEXT}_s(s, a_s) \cap C \neq \emptyset\} \quad (7.16) \end{aligned}$$

7.2.1 Weak Adversarial Precomponents

The weak adversarial precomponent consists of an ordinary weak precomponent pruned for unfair states. The precomponent function is shown in Figure 7.2. Let WEAKADVERSARIAL denote the NDP algorithm using the weak adversarial precomponent. Since the pruning of unfair states makes the SSAs in the precomponent robust for any non-empty environment plan, it can be shown that WEAKADVERSARIAL is sound, complete, and terminating.

function PRECOMPWA(C)
1 $wSA \leftarrow \text{PREIMGSSA}(C) \setminus C \times Act_s$
2 $waSA \leftarrow wSA \cap (\text{FAIRSTATES}(wSA, C) \times Act_s)$
3 **return** $waSA$

Figure 7.2: The weak adversarial precomponent function.

[2]When applying this algorithm for adversarial planning, the function STATES is substituted with the function STATES_s defined above.

Theorem 7.2 (Correctness of WeakAdversarial) *The* WEAKADVERSARIAL *planning algorithm is correct. The algorithm returns "no solution exists" iff no solution exists, otherwise it returns a valid solution.*

Proof. This follows from the soundness, completeness, and termination theorems of WEAKADVERSARIAL proven in Appendix B. □

A guided version of WEAKADVERSARIAL can be defined by using an approach similar to GUIDEDWEAK.

7.2.2 Strong Cyclic Adversarial Precomponents

Similarly to the strong cyclic precomponent, the strong cyclic adversarial precomponent is computed by iteratively expanding a candidate set and trying to show that it contains a valid precomponent. The precomponent function is shown in Figure 7.3. The main difference between the strong cyclic adversarial precomponent function and the strong cyclic precomponent function is that the auxiliary function SCAPLANAUX prunes unfair states from the precomponent instead of only unconnected states. This is done by iteratively computing the set of fair states in the precomponent starting from the covered states C. The computation also removes all unconnected states. Let STRONGCYCLICADVERSARIAL denote the NDP algorithm using the strong cyclic adversarial precomponent. It can be shown that STRONGCYCLICADVERSARIAL is sound, complete, and terminating.

Theorem 7.3 (Correctness of StrongCyclicAdversarial) *The* STRONGCYCLICADVERSARIAL *planning algorithm is correct. The algorithm returns "no solution exists" iff no solution exists, otherwise it returns a valid solution.*

Proof. This follows from the soundness, completeness, and termination theorems of STRONGCYCLICADVERSARIAL proven in Appendix B. □

A guided version of STRONGCYCLICADVERSARIAL can be defined by using an approach similar to GUIDEDSTRONGCYCLIC.

Example 7.5 Consider the strong cyclic adversarial precomponent computed from the goal state G of the adversarial planning problem introduced in Example 7.1. The first candidate precomponent is shown in Figure 7.4(a). Action $-s$ would have to be pruned from U since it has an outgoing transition. The pruned candidate is shown in Figure 7.4(b). Now there is no action leading to G in U when the environment chooses $+e$. U has become unfair and must be pruned from the candidate. The resulting candidate is shown in Figure 7.4(c). Since the remaining candidate is non-empty and no further state-action pairs need to be pruned, a non-empty strong cyclic adversarial precomponent has been found. ♢

function $\textsc{PreCompSCA}(C)$
$wSA \leftarrow \emptyset$
repeat
  $OldwSA \leftarrow wSA$
  $wSA \leftarrow \textsc{PreImgSSA}(C \cup \textsc{States}_s(wSA)) \setminus C \times Act_s$
  $SCA \leftarrow \textsc{SCAPlanAux}(wSA, C)$
until $SCA \neq \emptyset \vee wSA = OldwSA$
return SCA

function $\textsc{SCAPlanAux}(startSA, C)$
$SA \leftarrow startSA$
repeat
  $OldSA \leftarrow SA$
  $SA \leftarrow \textsc{PruneOutgoing}(SA, C)$
  $SA \leftarrow \textsc{PruneUnfair}(SA, C)$
until $SA = OldSA$
return SA

function $\textsc{PruneOutgoing}(SA, C)$
$NewSA \leftarrow SA \setminus \textsc{PreImgSSA}(\overline{C \cup \textsc{States}_s(SA)})$
return $NewSA$

function $\textsc{PruneUnfair}(SA, C)$
$NewSA \leftarrow \emptyset$
repeat
  $OldSA \leftarrow NewSA$
  $NewSA \leftarrow SA \cap \textsc{FairStates}(SA, C \cup \textsc{States}_s(NewSA)) \times Act_s$
until $NewSA = OldSA$
return $NewSA$

Figure 7.3: The strong cyclic adversarial precomponent function.

7.3 Action Selection Strategies

A strong cyclic adversarial plan guarantees that no intelligent environment can choose a plan that forces executions to cycle forever without ever reaching a goal state. In principle, though, infinite paths never reaching a goal state can still be produced by a system that "keeps losing" to the environment. However, by assuming the system selects randomly

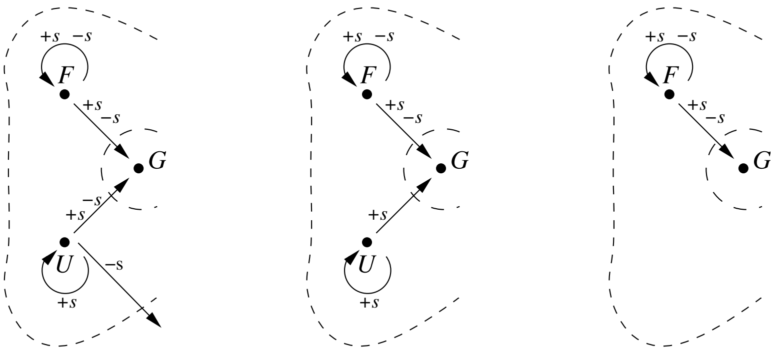


Figure 7.4: (a) The first candidate of PRECOMPSCA(G), for the problem shown in Figure 7.1(a). (b) The candidate pruned for actions with outgoing transitions. (c) The remaining candidate pruned for unfair states. Since no further SSAs are pruned, this is the strong cyclic adversarial precomponent returned by PRECOMPSCA(G).

between actions in its plan, we can show that the probability of producing such execution paths is zero.

Theorem 7.4 (Termination of Strong Cyclic Adversarial) *By choosing actions randomly from a strong cyclic adversarial plan π_s produced by* STRONGCYCLICADVERSARIAL *given the adversarial planning problem $\mathcal{P} = \langle \mathcal{D}, s_0,$ $G\rangle$, any execution path will eventually reach a goal state.*[3]

Proof. Since all unfair states and actions with transitions leading out of the states covered by π_s have been removed, all the visited states of an execution path will be fair and covered by the plan. Assume without loss of generality that n strong cyclic adversarial precomponents were computed in order to generate π_s. Due to the definition of precomponent functions, we can then partition the set of states covered by π_s into $n+1$ ordered subsets $C_n, \cdots, C_0$ where $s_0 \in C_n$, $C_0 = G$, and C_i for $0 < i \leq n$ contains the states covered by precomponent i. Consider an arbitrary subset C_i. Assume that there were m iterations of the repeat loop in the last call to PRUNEUNFAIR when computing precomponent i. We can then subpartition C_i into m ordered subsets $C_{i,m}, \cdots, C_{i,1}$ where $C_{i,j}$ contains the states of the SSAs added to $NewSA$ in iteration j of PRUNEUNFAIR. Due to the definition of FAIRSTATES, we have that the states in $C_{i,j}$ are fair with respect to π_s and the states C given by

$$C = \bigcup_{k=1}^{j-1} C_{i,j} \cup \bigcup_{k=0}^{i-1} C_i.$$

[3]It is likely that the theorem holds for any strong cyclic adversarial plan satisfying Definition 7.9. However, the proof must be strengthened to show this.

By flattening the hierarchical ordering of the partitions $C_n, \cdots, C_0$ and their subpartitions, we can assume without loss of generality that we get the ordered partitioning $L_T, \cdots, L_0$ where $L_0 = C_0$. Given that actions are selected uniformly in π_s, the fairness between the states in the levels guarantees that there is a non-zero probability to transition to a state in $L_{i-1}, \cdots, L_0$ from any state in L_i. Consequently, an execution path only reaching states covered by π_s will eventually reach a state in L_0. $\square$

For weak adversarial plans, it is impossible to guarantee that a goal state eventually is reached since an execution path may reach a dead end. On the other hand, by selecting actions randomly from a weak adversarial plan there is a non-zero probability of reaching a goal state.

Theorem 7.5 (Progress of Weak Adversarial) *By choosing actions randomly from a weak adversarial plan π_s poduced by* WEAKADVERSARIAL *given the adversarial planning problem $\mathcal{P} = \langle \mathcal{D}, s_0, G \rangle$, there is a non-zero probability of eventually reaching the goal.*[4]

Proof. Assume without loss of generality that n weak adversarial precomponents were computed in order to generate π_s. Due to the definition of precomponent functions, we can then partition the set of states covered by π_s into a $n+1$ ordered subsets $C_n, \cdots, C_0$ where $s_0 \in C_n$, $C_0 = G$ and C_i for $0 < i \leq n$ contains the states covered by precomponent i. Consider an arbitrary subset C_i. Due to the definition of FAIRSTATES, we have that the states in C_i are fair with respect to π_s and the states C given by

$$C = \bigcup_{k=0}^{i-1} C_i.$$

Thus, given that actions are selected uniformly in π_s, we have a non-zero probability to transition to a state in $C_{i-1}, \cdots, C_0$ from any state in C_i. Consequently, there is a non-zero probability of an execution path starting in s_0 and reaching a goal state in G. $\square$

7.4 Experimental Evaluation

The performance of WEAKADVERSARIAL and STRONGCYCLICADVERSARIAL has been evaluated in two domains. The first of these is a parameterized version of the example domain shown in Figure 7.1. The second is a grid world with a hunter and prey. Due to time limitations and the lack of benchmark problems, the guided versions of the algorithms have not been studied. However, we expect performance improvements between the blind and

[4]It is likely that the theorem holds for any weak adversarial plan satisfying Definition 7.9. However, the proof must be strengthened to show this.

guided versions of the adversarial algorithms that are similar to the performance improvements obtained for the blind and guided version of the weak and strong cyclic algorithms in Section 5.3.

All experiments are carried out using the BIFROST 0.7 search engine and the experimental setting described in Appendix A. The problems of both domains have been described in NADL$^+$. As usual, we use n to denote the number of BDD-nodes allocated to represent the shared BDD, and c to denote the number of BDD nodes allocated to represent BDDs in the operator caches used to implement dynamic programming. Total CPU time is measured in seconds and includes time spent on allocating memory for the BDD package and parsing the problem description.

7.4.1 Parameterized Example Domain

The parameterized example domain considers a system and environment actions $\{+s, -s, l\}$ and $\{+e, -e\}$, respectively. The domain is shown in Figure 7.5. The initial state is $s_0 = I$ and the goal states are $G = \{g_1, g_2\}$. Progress toward the goal states is made if the sign of the two actions in the joint action are different. At any time, the system can cause a switch from the lower to the upper row of states by executing l. In the upper row, the system can execute only $+s$. Thus, in these states an adversarial environment can prevent further progress by always executing $+e$. Figure 7.6 shows the total CPU time and the size

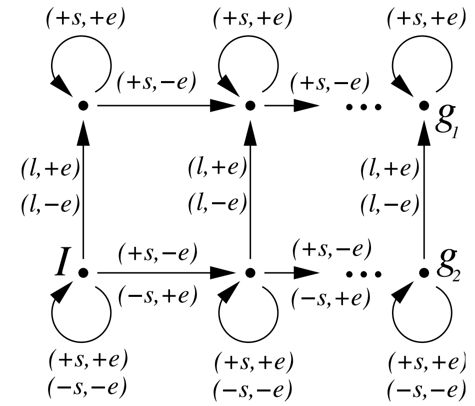


Figure 7.5: The generalized example domain shown in Figure 7.1(a).

of the produced plans of the ordinary weak algorithm compared to weak adversarial algorithm and the ordinary strong cyclic algorithm compared to the strong cyclic adversarial algorithm. The BDD variable ordering was identical in all of these experiments. For each experiment, the BDD package was initialized with $n = 1M$ and $c = 700K$. The total time used for memory allocation was 0.7 seconds. Due to the structure of the domain, the length

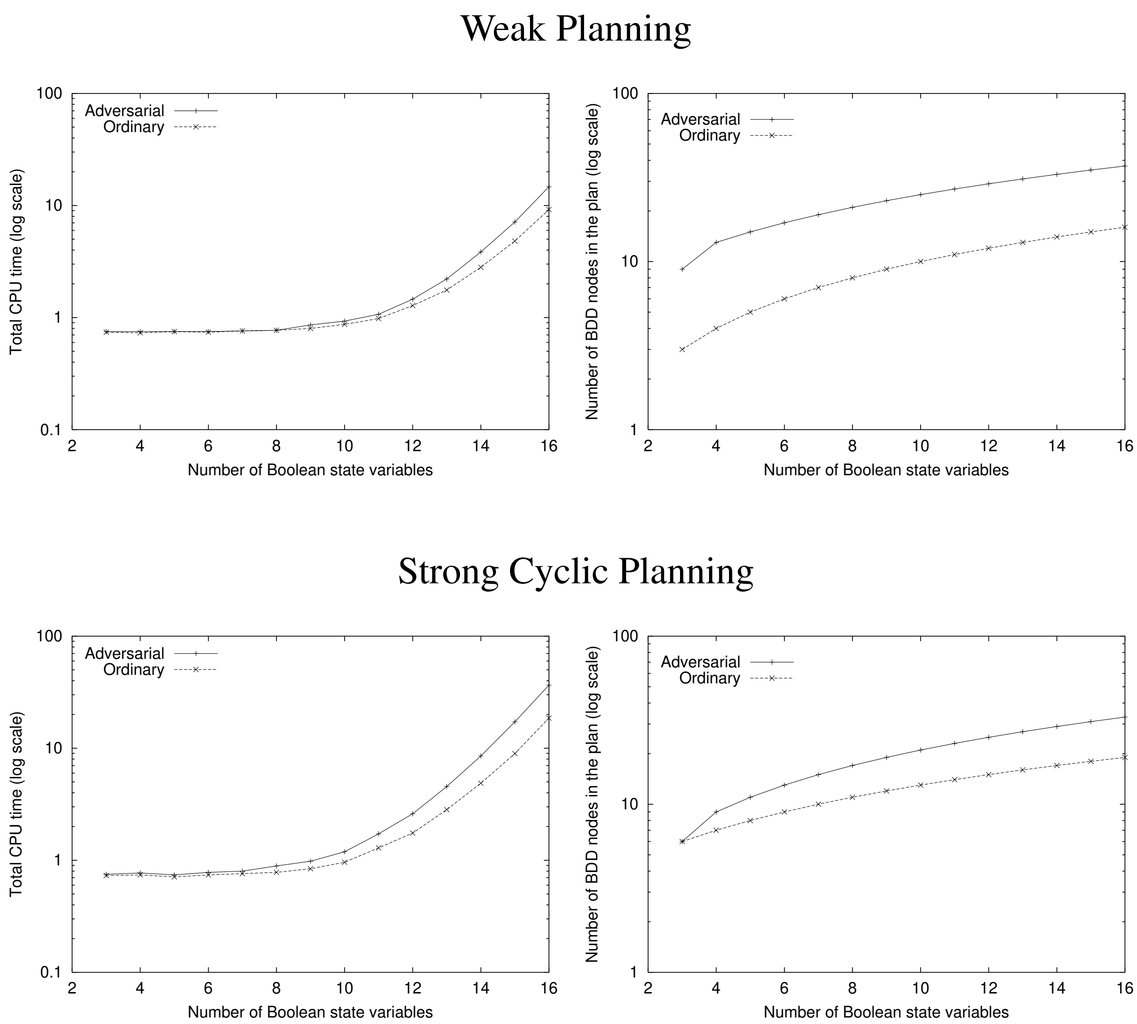


Figure 7.6: Results of the parameterized example domain.

of a shortest path between the initial state and one of the goal states grows linearly with the number of states. Since the four algorithms must compute at least one preimage for each step in a shortest length path between the initial state and one of the goal states, their complexity is at least exponential in the number of Boolean state variables. The experimental results seem to confirm this. In this domain, there only is a small overhead of generating adversarial plans compared to non-adversarial plans. The quality of the produced plans, however, is very different. For instance, the strong cyclic adversarial plans consider executing only $-s$ and $+s$, while the strong cyclic plans consider all applicable actions. The strong cyclic adversarial plan is guaranteed to achieve the goal. In contrast, the probability of achieving the goal in the worst case for the strong cyclic plan is less than $(\frac{2}{3})^{N/2-1}$, where N is the number of states in the domain. Thus, for an adversarial environment the probability of reaching the goal with a strong cyclic plan is practically zero, even for small instances of the problem.

7.4.2 Hunter and Prey Domain

The hunter and prey domain consists of a hunter and prey agent moving on a chess board. Initially, the hunter is at the lower left position of the board and the prey is at the upper right. The initial state of the game is shown in Figure 7.7. The task of the hunter is to catch

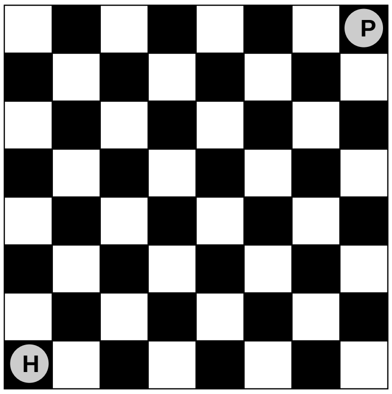


Figure 7.7: The Hunter and Prey domain.

the prey. This happens if the hunter and prey at some point are at the same position. The hunter and prey move simultaneously. They are not aware of each others moves before both moves are carried out. In each step, they can either stay at the spot or move like a king in chess. However, if the prey reaches the lower left corner position, it may change the moves of the hunter to that of a bishop (making single step moves). This has a dramatic impact on the game, since the hunter then can move only on positions with the same color. Thus, to avoid the hunter, the prey just have to stay at positions with opposite color. A strong cyclic adversarial plan therefore only exists if it is possible for the hunter to find a plan that guarantees that the prey never gets to the lower left corner. A strong cyclic plan, on the other hand, does not differentiate between whether the hunter moves like a chess King or a Bishop. In both cases, a “friendly” prey can be caught.

We consider a parameterized version of the domain with the size of the chess board ranging from 8×8 to 512×512. For the 8×8 board, we need 3 Boolean variables to represent the vertical and horizontal location. This gives $4 * 3 = 12$ Boolean variables. Similarly, for the 512×512 board, we need $4 * 9 = 36$ Boolean variables. Figure 7.8 shows the total CPU time and the size of the plans produced by the ordinary weak algorithm compared to weak adversarial algorithm and the ordinary strong cyclic algorithm compared to the strong cyclic adversarial algorithm. For each experiment, the BDD package was initialized with $n = 12M$ and $c = 500K$. The total time used for memory allocation was 2.8 seconds. In this domain both weak and strong cyclic adversarial plans are larger and

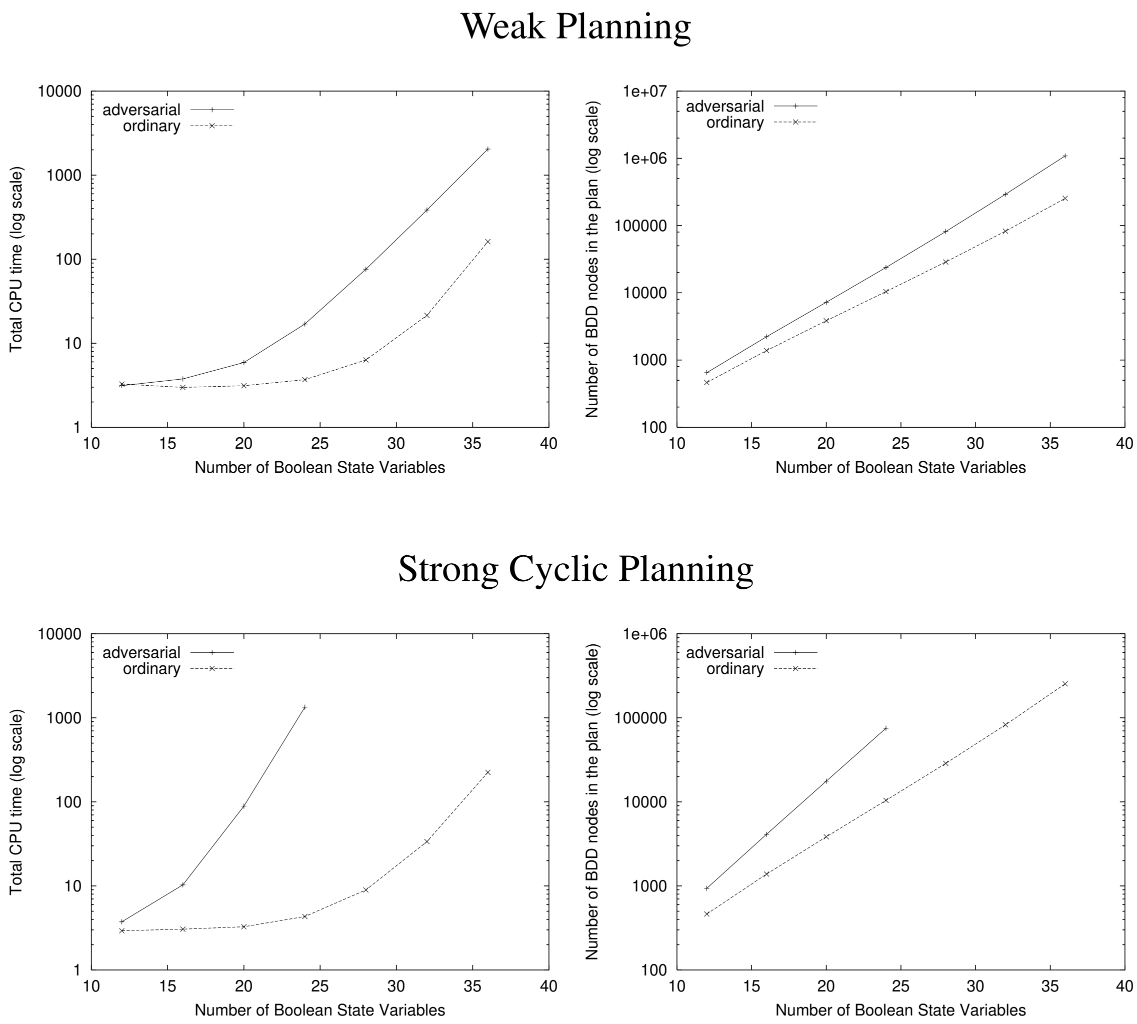


Figure 7.8: Results of the hunter and prey domain.

take substantially longer time to generate than ordinary plans. The strong cyclic adversarial algorithm spends more than 4000 seconds for problems with 28 Boolean state variables or more. However, as discussed above, it is non-trivial to determine whether there exists a strategy of the hunter that guarantees that the prey never succeeds in reaching the lower left corner. Thus, we may expect these plans to be computationally harder than strong cyclic plans. This interpretation is supported by the size of the plans. The adversarial plans are substantially larger than the ordinary plans. We have not analysed the strong cyclic adversarial plans in detail, but they at least must fulfill that the prey never gets closer to the lower left corner than the hunter. The hunter starts at the lower left corner and can therefore ensure that the prey must "risk its life" even by getting to a location with the same distance to the lower left corner as the hunter. Since the hunter still has control over the game in this situation, it has a positive chance of winning. Thus, a strong cyclic adversarial plan exists.

7.5 Conclusion

The two major design goals of the weak and strong cyclic adversarial planning algorithms is that they are correct and efficient. With respect to correctness, the strong cyclic adversarial algorithm has been chosen to closely match the strong cyclic algorithm such that a similar proof strategy can be applied. With respect to efficiency, three major choices have been made. First of all, we use a BDD-based implementation. Second, as for the strong cyclic algorithm, we build up a strong cyclic adversarial plan incrementally from the goal states. Alternatively, the algorithm could iteratively prune a largest possible plan as the algorithm suggested in [43]. However, this approach seems less efficient. Finally, guided versions of the algorithms can be defined using non-deterministic state-set branching. An interesting direction for future work is to combine fault tolerant and adversarial planning and to consider an explicit set of goal states of the environment agent [23].

7.6 Summary

In this chapter, we have introduced a new framework called adversarial planning to address domains where non-determinism is caused by simultaneous actions of a controllable system and an uncontrollable environment. We have shown that the usual abstraction of environment actions may lead to solutions where an adversarial environment may cause execution paths never to reach a goal state. This can be avoided by pruning states from the plans where the local "game" between the system and environment is unfair. We introduce adversarial versions of the weak and strong cyclic non-deterministic planning algorithms and show that they are sound, complete, and terminating. The experimental evaluation shows that adversarial plans may be harder to produce than ordinary non-deterministic plans. However, this is to be expected since they often represent more complex control strategies.

Chapter 8

Related Work

The discussion of related work is divided into five sections corresponding to the five main contributions of the thesis. Section 8.1 describes work related to BDD-based deterministic planning and state-set branching in Artificial Intelligence (AI) and formal verification. Section 8.2 first discusses alternative approaches to non-deterministic planning in AI, automata theory, game theory, and Discrete Event System (DES) control theory and then focuses on work closely related to non-deterministic state-set branching. Section 8.3 presents work related to fault tolerant planning within DES control theory and AI, and Section 8.4 describes work related to adversarial planning developed in automata theory, game theory, AI, and formal verification. Finally, Section 8.5 reviews related work on planning languages developed in AI, formal verification, and DES control theory.

8.1 Deterministic Planning and Heuristic Search

An interesting fact is that even the earliest planning systems were using a symbolic representation of the state space. The most popular representation is STRIPS where a search state is a set of facts that are true in the set of domain states, the search state represents (a detailed description of STRIPS planning is given in Section 3.1). *Progression* planners search forward in the fact space. Since they start from a set of facts representing a single initial state, each search state corresponds to a single state. Hence, for progression planners, no space savings are obtained with the symbolic representation compared to an explicit representation. *Regression* planners, on the other hand, search backward in the fact space. Since this search starts from a single set of facts representing a set of goal states, each search state may represent several domain states. Thus, regression planners may benefit from the symbolic state representation.

A wide range of planning systems and search techniques have been developed within the STRIPS framework. These planning systems are often referred to as *classical planners* and roughly fall in three classes: *state space planners* (e.g. PRODIGY [133]), *plan space planners* (e.g. SNLP [116] and UCPOP [163]) and *hierarchical planners* (e.g. SIPE [171]). The probably most advanced state space planner is PRODIGY. PRODIGY performs a bidirectional search in the fact space guided by *search control rules*, *means-end analysis* and *sub-goaling* [126]. Plan space planners, on the other hand, carries out a search in a space of possible plans using the *least commitment principle* where orderings between actions only are introduced if the actions are causally linked or interfere. This may lead to better performance in some domains [8]. However, plan space planners commit to causal links in much the same way that state space planners commit to step ordering. In general, they do not outperform state space planners [165]. Hierarchical planners such as SIPE use hierarchical task networks (HTNs) to apply abstraction in the search. First, a solution is found at an abstract level which then is refined to a concrete plan.

The scalability of classical STRIPS planning was substantially improved by the introduction of GRAPHPLAN [18] that avoids the state space explosion problem by using a planning graph to guide the search. Graph planners [18, 114, 104] use a two step approach. In the first step, the planning graph is generated. The planning graph consists of alternating action and state layers and keeps track of the interferences between actions and states resulting in a compact representation of the reachable states. In the second step, a plan is extracted from the planning graph by a backwards search. Graph planners relax optimality constraints by only finding parallel optimal plans, i.e., plans with shortest length assuming that actions can be applied concurrently in each step.

One of the current trends in automated planning is to reduce planning to other problems. SAT planners, like SATPLAN [99], encode a planning problem as a satisfiability problem of a Boolean expression stating goal achievement within a certain number of steps. Using binary search, this approach can be optimal, but better results have been obtained using GRAPHPLAN's parallel relaxation by encoding goal achievement of the planning graph as a SAT problem (BLACKBOX,[100]). Planning as satisfiability, however, suffers from the fact that the number of Boolean variables grows linearly with the plan length. In addition, the clauses of a planning problem form long dependency chains (corresponding to plans) that are known to be a worst case structure for SAT checkers [9].

A few experiments have also been carried out reducing planning to integer programming [19, 101]. This approach works well if a considerable part of the planning problem involves numerical constraints. Good performance, however, has not been obtained for purely combinatorial problems.

The first application of BDDs for deterministic planning was based on reduction of planning to symbolic model checking (deterministic MBP, [33]), where the plan corresponds to a counter example of a verified property. The approach has been shown to be competitive with GRAPHPLAN and SATPLAN in several classical domains. More recent approaches are MIPS 1.0, DOP, BDDPLAN and PROPPLAN [54, 92, 80, 59]. All of these planners rely on blind BDD-based breadth-first search. MIPS 1.0 and DOP uses a specialized preprocessing of domains to find compact Boolean state encodings [50]. Both planners apply bidirectional search from the initial and goal states. BDDPLAN and PROPPLAN are more simple BDD-based planners without domain preprocessing and have poor performance compared to MIPS 1.0 and DOP.

Blind BDD-based search is currently one of the most efficient approaches for finding optimal plans in deterministic domains [86]. However, when optimality constraints are relaxed, the currently most efficient approach for the benchmark problems considered at the AIPS planning competitions [113, 4, 115] are pure heuristic planners like HSP, FF, and ALTALT [20, 176, 77]. However, it has been shown that the HSPr derived heuristics used by these planners have no plateaus in the competition domains making simple hill climbing a sufficiently strong search approach to find a solution [79]. It seems implausible that such strong heuristics are easy to define for a larger set of benchmark problems.

State-Set Branching

As far as we know, state-set branching is the first general framework for combining heuristic search and BDD-based search. All previous work has been restricted to particular algorithms. BDD-based heuristic search has been investigated independently in symbolic model checking and AI. The pioneering work is in symbolic model checking where heuristic search has been used to falsify design invariants by finding error traces. Yuan et al. [180] studies a bidirectional search algorithm pruning frontier states according to their minimum Hamming distance to error states. BDDs representing Hamming distance equivalence classes are precomputed and conjoined with BDDs representing the search frontier during search. Yang and Dill [179] also consider minimum Hamming distance as heuristic function in an ordinary pure heuristic search algorithm. They develop a specialized BDD operation for splitting a set of states according to their minimum Hamming distance to a set of error states. The operation is efficient. Its complexity is linear with the size of the BDD representing the error states. However, it is unclear how such an operation can be generalized to other heuristic functions. In addition, this approach finds next states and splits them according to their Hamming distance to the goal states in two separate phases where the first phase is as complex as the single expansion phase used by state-set branching.

In general, heuristic BDD-based search has received little attention in symbolic model checking. There may be several reasons for this

1. *Culture*. Heuristics and heuristic search has mainly been studied in AI,

2. *Lack of Efficient Heuristics*. Symbolic model checking problems often consider sequential circuits where all state variables are changed in each step. The diameter of the transition graph may be too low for an efficient search heuristic to exist [144],

3. *A Different Problem*. Planning problems are inherently different from verification problems. In order to verify a system all reachable states must be explored. However, in order to solve a planning problem only a single path from the initial state to the goal state needs to be found.

An important exception to the second statement is verification of asynchronous systems. In the SPIN validater [81] and JAVA PathFinder [166], several heuristics have been studied to guide the search toward counter examples (e.g.,[52, 69]). In addition, a number of heuristic methods have been developed to guide the exploration of a CTL formula in order to reduce the complexity of the model checking problem [17, 82].

In AI, an implementation of A* called BDDA* was developed by Edelkamp and Reffel [53]. BDDA* can use any heuristic function and has been applied to planning as well as model checking [144]. Edelkamp later describes a more general implementation of BDDA* not assuming unit-cost transitions and with cycle detection for monotonic heuristic functions [48]. Both of these versions of BDDA*, however, are fairly direct implementations of A* with BDDs that imitates the usual explicit application of the heuristic function via complex symbolic arithmetic. Our experimental results show that the successor state computation of BDDA* scales poorly. For this reason a major philosophy in the design of state-set branching has been to avoid arithmetic operations at the BDD level. An ADD-based implementation of A* called ADDA* [74] has been developed after the first publication of state-set branching. ADDs [6] generalize BDDs to finite valued functions. ADDA* is similar to BDDA* but implements cycle detection for general heuristic functions. The ADD may handle arithmetic computations more efficiently than the BDD [168]. However, ADDA* has not successfully been shown to have better performance than BDDA* [74].

The high performance of state-set branching is achieved by the branching partitioning that combines an efficient partitioned image and preimage computation with a propagation of search node information from parent to child states. The philosophy of state-set branching is that the information represented by BDDs must be semantically closely related in order for the BDD operations to work efficiently. Hence, in contrast to BDDA* and ADDA*, we separate the representation of information used by the search algorithm from the representation of states and transitions and only employ BDDs to encode the latter.

To our knowledge, this idea is genuinely new. We have not been able to find any previous work in either AI, control theory, automata theory, and formal verification that use a transition relation partitioning for propagating any kind of state information. There seems to be several circumstances that may explain why this particular stone never has been turned before. In AI, the main reason seems to be that the amount of work involving BDDs still is very limited. The only BDD-based classical heuristic search algorithms in AI are BDDA* and ADDA* and these algorithms seem to rely on a quite different design philosophy where as much information as possible has been pushed to the BDD level. In control theory and automata theory, symbolic controller synthesis been suggested but not sufficiently investigated. As far as we know, there has not been any work on guided synthesis algorithms. A relevant area to expect previous work, is formal verification. There is a large body of work on reducing the complexity of BDD-based search. In addition, it was within this area that the first guided BDD-based search algorithms were invented. There seems to be two reasons why an approach similar to state-set branching has not been considered. First, even though the state-set branching approach covers both asynchronous and synchronous systems, it is more obvious to consider for an asynchronous system where the transition relation can be efficiently encoded by a disjunctive partitioning. However, in formal verification, most work on symbolic model checking considers synchronous systems. Second, as discussed previously, heuristic search is often inefficient for synchronous systems with a low transition graph diameter. Thus, only a limited amount of work has gone in this direction.

8.2 Non-Deterministic Planning

Non-deterministic planning in different disguises has been studied in AI, automata theory, game theory, and DES control theory. The classical approach to non-deterministic planning in AI is *conditional planning*. Conditional actions were first studied in WARPLAN-C [167]. Modern conditional planners includes (CNLP, [134], C-BURIDAN [46], and SGP [169]). CNLP is an extension of the partial order planner SNLP. It handles non-determinism by constructing a conditional plan that accounts for each possible situation or contingency that could arise. At execution time it is determined which part of the plan to execute by performing sensing actions that are included in the plan to test for the appropriate conditions. The returned plan is a finite tree where each branch is a sensing action. C-BURIDAN combines conditional and probabilistic planning. A sensing action can be inserted in the plan to increase the success probability. Branches can be rejoined such that the resulting plan is more compactly represented as a DAG. SGP descends from GRAPHPLAN. It has been shown to outperform any of the previous planners obtained as extensions to classical planners [169]. The most important limitation of conditional planning compared to non-deterministic planning as defined in this thesis is that conditional plans are finite and may grow exponentially with the number of unknown facts. A performance comparison between the BDD-based non-deterministic planning system MBP [34] and SGP on the Omelet problem from the SGP distribution shows that MBP scales much better than SGP on this problem. Conditional plans can also be generated by QBFPLAN [145]. QBFPLAN is a generalization of the SATPLAN approach to the case of planning in non-deterministic domains. The user must provide the number of control points and observations in the plan. This can provide a significant limitation of the search space. However, in the Chain domain provided with the QBFPLAN distribution, MBP outperforms it severely [34]. These experimental results indicate that BDD-based non-deterministic planning is one of the most efficient approaches to conditional planning. In particular, BDD-based non-deterministic planning seems to be least sensitive to the amount of non-determinism in the domain. On the other hand, if non-determinism is sparse, the planning graph approach employed by SGP may be more efficient.

Universal planning [154] is the non-deterministic planning approach closest related to the approach investigated in this thesis. The main difference is that we model non-determinism explicitly. Instead, the original idea in universal planning is to cover every domain state in order to make the plan robust to non-determinism (e.g., caused by failures or simultaneous activity). A major challenge is to represent universal plans compactly. It has been shown that even for a flexible circuit representation of universal plans in domains with n Boolean state variables, the fraction of randomly chosen universal plans with polynomial size in n decreases exponentially with n [65]. [1] However, universal plans encountered in practice are normally far from randomly distributed. Often real-world planning problems and their universal plan solutions are regularly structured. A primary objective is therefore to develop efficient techniques for exploiting such structure. It is exactly the ability of BDDs to capture structure of many Boolean functions often met in practice that makes them attractive for representing universal plans.

The first BDD-based universal planning system was MBP [36, 37]. The approach used by MBP was further explored in UMOP [93] which together with MBP is the only current BDD-based universal planning system. An alternative approach to universal planning is SIMPLAN [98]. SIMPLAN generates a plan from a forward search that may be guided by an LTL control rule formula [5]. It can synthesize plans for extended goals in Linear Temporal Logic (LTL) [139]. This includes strong plans, but not strong cyclic plans, which only can be expressed in CTL. A head-to-head comparison between MBP and SIMPLAN in a robot delivery domain provided in the SIMPLAN distribution shows that SIMPLAN is

[1] There is nothing new in this result. Ginsberg's circuit has the same fate as any other known representation of Boolean functions [121].

very sensitive to non-determinism and is outperformed by MBP even when applying search control rules. MBP has later been extended to handle temporally extended goals given as a CTL formula [138].

Non-deterministic planning as defined in this thesis does not involve transition probabilities. While transition probabilities can provide useful information in some domains, there are domains where modeling transition probabilities is hard in practice due to the lack of statistical data. In addition, we may expect the computational complexity of *probabilistic planning* to be higher than non-deterministic planning due to the more expressive domain model. The collection of probabilistic planners include DRIPS [70] and BURIDAN [110]. DRIPS decomposes operators of an abstraction hierarchy of operators in order to find plans with maximum expected utility. BURIDAN is derived from SNLP and produces plans that meet a threshold probability. The produced plans are finite and acyclic and are therefore in general insufficient to guarantee goal achievement.

Until now, we have only considered planning approaches where a plan is produced prior to execution. An alternative approach is to perform planning interleaved or in parallel with execution. This can either be done by monitoring the plan execution and re-plan whenever an action fails or select an action in each step of the execution. *Plan monitoring* and *re-planning* have been widely used in non-deterministic robotic domains (e.g., [62, 172, 71]). Action selection planners can be based on real-time heuristic search algorithms like MIN-MAX LRTA* [105, 106]. The MIN-MAX LRTA* search algorithm can generate suboptimal plans in non-deterministic domains through a search and execution iteration. The search is based on a heuristic goal distance function that must be provided for a specific problem. The ASP algorithm [21] uses a similar approach based on the HSP heuristic [20]. In contrast to MIN-MAX LRTA*, ASP does not assume a non-deterministic environment, but is robust to non-determinism caused by action perturbations (i.e., that another action than the planned action is chosen with some probability). In general, planners interleaving planning and execution are incomplete because acting on an partial plan can make the goal unachievable. However, they are often efficient in robotics domains where most actions are reversible.

Non-deterministic planning, as defined in this thesis, is assuming full observability of the states. Another extreme is to assume that the states are unobservable and generate *conformant plans* [68, 169]. A conformant plan is a sequence of actions that leads to the goal independently of non-determinism in the domain. A BDD-based approach to conformant planning has been studied in the MBP planning framework [35] as well as an approach to non-deterministic planning in domains with partially observable states [13]. In the latter work, heuristics have been applied to guide the expansion of an AND-OR graph where nodes are BDDs representing sets of belief states [12].

We now turn to discuss approaches to non-deterministic planning developed outside of the field of automated planning. Reinforcement Learning (RL) [161] can be regarded as non-deterministic planning. In RL the goal is represented by a reward function in a Markov Decision Process (MDP) model of the domain. A non-deterministic plan solving the problem is a policy mapping states to actions that maximizes the expected reward. The policy can either be represented explicitly in a table or implicitly by a function (e.g., a neural network). The major limitation of RL is its ability to scale. If states are represented explicitly only very small problem instances can be solved. Function approximation methods may be applied to obtain an implicit representation of the domain. However, this may compromise the convergence of the value-iteration methods used to find policies [24]. Symbolic approaches have been applied to RL. SPUDD [76] uses the Algebraic Decision Diagram (ADD) [6] to represent value functions and policies. The value-iteration computation of SPUDD is implemented via ADD manipulations. Substantial performance gains may be obtained with SPUDD compared to ordinary RL methods. Compared to BDD-based non-deterministic planning, however, SPUDD is limited by the fact that it must represent a possibly fast growing set of different values of the value function.

The strong algorithm in different disguises has been discovered independently in automata theory [3], automated planning [36, 37] and game theory [43]. In addition, symbolic methods for supervisory controller synthesis that in principle can be used to synthesize weak, weak adversarial, strong cyclic, strong cyclic adversarial and strong plans were suggested as early as in 1992 [78]. However, these specific algorithms have, as far as we know, not been described in the DES control theory literature. We are also not aware of any work in DES control theory that studies the efficiency aspect of symbolic controller synthesis.

Non-Deterministic State-Set Branching

Non-deterministic state-set branching is to our knowledge the first attempt to guide a BDD-based search for a non-deterministic plan as defined in this thesis. We have not been able to find any previous work of this kind in automated planning, automata theory, DES control theory, and game theory. The closest work is the symbolic LAO* algorithm [57] used to solve MDPs. However, this algorithm can not be applied to problems without transition probabilities.

8.3 Fault Tolerant Planning

Most of the non-deterministic planning approaches discussed in the previous section focuses on domains where failure is a key aspect. This is also the case for the large body of work in AI on fault diagnosis (e.g., [102, 73, 155, 45]). However, work explicitly representing and reasoning about success and failure effects of actions is very limited. The ELMER system [117] uses error transitions from abstract actions to detect and recover from failures. In the Procedural Reasoning System (PRS) [62], the procedure descriptions defines the effect of successful and unsuccessful execution of a procedure. Similarly, the Reactive Model Based Programming Language (RMPL) [174] and its underlying executor Titan can handle faults at runtime. The approach, however, does not involve computing a fault tolerant plan. The MRG [67] planning language explicitly models failure effects. However, this work does not include planning algorithms for generating fault tolerant plans. To our knowledge, the n-fault tolerant planning algorithms introduced in this thesis are the first automated planning algorithms for generating fault tolerant plans given a description of the domain that explicitly represents failure effects of actions.

Similarly to AI, there has been substantial amount of work on fault diagnosis in DES control theory. This work has mainly focused on analysing event sequences in order to determine if a fault has happened, and if so, which kind of fault [150, 151, 152, 159]. However, there has also been a considerable amount of work on fault models. These models can be characterized as either *transition based* or *state based*. Most work (e.g., [31, 32, 38]) use the transition based model and regard *faults* as unexpected changes in a system that tends to degrade the overall system performance rather than causing a total breakdown. The term *failure* suggests a complete breakdown of a system component or function. The transition based model is also used in supervisory control [142] where faults usually are considered uncontrollable events [7, 32]. Within this frame, an approach to fault tolerant control has been considered that is closely related to n-fault tolerant planning. The work in [135], specifies fault tolerance for mission critical systems. A *masking fault tolerant system* can recover from any fault. A *t-fault tolerant system* can recover from up to t faults occurring during its life time. The system is modeled by an automaton with start states, but no goal states. In addition, no algorithms or theory for controller synthesis are provided.

The state based models usually divides the state space into ranges of operation of some system (e.g., “normal operation range”, “admissible error range”, and “non-admissible error range” [103], or “good” and “bad” states [128]). In Özveren’s work [128], *Stability* is defined to be to visit the good states infinitely often. Thus, a controller is *stable* if it from any reachable bad state can force a trajectory that in a finite number of steps reaches the good states. *Stabilizability* is defined to choosing state feedback such that the closed loop system is stable. A related approach [129] defines *Lyapunov stability* of a class of DES.

Consider a set of states X_m that are invariant in the plant. That is, any execution starting in any state in X_m stays within X_m. X_m is stable in the sense of Lyapunov if for any $\epsilon > 0$ a max distance $\delta > 0$ (given by some metric) can be found such that any execution starting at a state within δ from X_m ends up in a state less than ϵ from X_m.

We are not aware of work in any other field than AI and DES control theory that reason explicitly about failures in order to automatically synthesize fault tolerant plans or fault tolerant discrete controllers.

8.4 Adversarial Planning

Adversarial planning is related to work in AI on negotiation (e.g.,[30, 181, 108]) and collaboration (e.g.,[63, 47, 83]) in multi-agent systems. The focus in this work, however, is more on establishing frameworks for describing these problems than developing efficient algorithms for solving them. In particular, the only previous BDD-based multi-agent planning system that we are aware of is UMOP [93] which is a predecessor to the work described in this thesis. Another direction of work in AI applies planning algorithms to search in a space of game states (e.g.,[170] Chess, and [157] Bridge). However, in contrast to the adversarial planning algorithms introduced in this thesis, these approaches do not consider complete solutions of the game. This is also not the case for game tree algorithms like ALPHA-BETA-MINIMAX [125].

Adversarial planning is related to game theory in the sense that both offer alternative approaches for generating policies for adversarial environments. For instance, we could enumerate all policies of the environment and the system creating an appropriate payoff matrix, and solve this as a *normal form matrix game* [127]. Alternatively, we could apply game tree algorithms and solve the planning problem as an *extensive form game* [127]. However, both of these approaches are intractable due to the exponential size of the matrix and game tree in the number of state variables. The game-theoretic framework that is closest related to adversarial planning is stochastic games. Stochastic games extend Markov decision processes to multiple agents. They are usually solved using value iteration algorithms that require exponential space in the number of state variables (see e.g. [156]). Function approximation techniques may be able to reduce the space requirements. However, it is still unclear how these methods can be applied without sacrificing the convergence properties of the value iteration algorithms. One of the advantages of BDD-based adversarial planning is to avoid such explicit representations.

It has been noted in automata theory that *winning strategies* in two player games correspond to strong plans and that such strategies can be computed symbolically using BDDs [3]. This is independent of whether the moves by the two players are simultaneous or interleaved. The non-deterministic model is strong enough to represent both situations. Thus, this early work in automata theory to some extend subsumes a later work in automated planning employing BDDs for two-player games with alternating moves [49].

Adversarial planning has been studied in formal verification in the form of *concurrent reachability games* [43, 2, 97]. A strategy of a player is a mapping from states to a probability distribution over a set of actions to apply in the state. A state s is *sure* if player 1 (the system) has a strategy so that for all strategies of player 2 (the environment), the game, if started in s, always reaches a set of target states (goal states). Hence, a state is sure, if player 1 has a strong plan for reaching the target states. A state s is *almost sure* if player 1 has a strategy so that for all strategies of player 2, the game, if started in s, reaches a target state with probability 1. Thus, a state is almost sure, if player 1 has a strong cyclic adversarial plan for reaching the target states. Finally, a state s is *positive* if player 1 has a strategy so that for all strategies of player 2, the game, if started in s has a positive probability of reaching the target states. This corresponds to a weak adversarial plan.

It is observed that the set of sure, almost sure, and positive states can be computed symbolically without representing probabilities. An algorithm similar to STRONGis given to compute sure states [43]. The algorithm for computing almost sure states is dual to STRONGCYCLICADVERSARIAL in the sense that it starts from all the states in the domain and the most general strategy of player 1 and then iteratively prunes states and actions from the strategy that can lead to states where player 2 can confine the game. Instead, STRONGCYCLICADVERSARIAL iteratively increments the set of almost sure states. The work in [43, 2, 97] is theoretical. There is no experimental evaluation of the approach. The primary goal of our work on adversarial planning is scalability. The incremental approach of STRONGCYCLICADVERSARIAL has been chosen because we believe this approach is more efficient even in its blind version. More importantly, however, this format of the algorithm makes it possible to apply search heuristics using non-deterministic state-set branching.

8.5 Planning Languages

Classical deterministic planning languages like STRIPS [58], ADL [132], and PDDL [118] represent domains in first order logic. Such representations can be encoded compactly with BDDs as described in Section 3.1.1, but it is more natural to use state variable representations as in NADL$^+$. Non-deterministic planning languages related to NADL$^+$ includes $\mathcal{AR}$ [66] and NuPDDL [137] that both are used as input languages to MBP [33]. The action description language $\mathcal{AR}$ can represent propositional and non-propositional fluents with finite domains. Actions may change the value of fluents non-deterministically. Compared to $\mathcal{AR}$, NADL$^+$ introduces numerical state variables, an explicit environment model, and an explicit representation of failure effects of actions. In addition, it includes features for defining transition costs and for propagating search information between states. The only prior planning language, we are aware of, that explicitly models action failure is MRG [67]. However, this language does not explicitly model the actions of an uncontrollable environment. NuPDDL descends from PDDL 2.1 [60] that can represent numeric-valued fluents and time. In addition, nuPDDL can model uncertainty in initial states and non-deterministic action effects. However, it has no constructs for explicitly describing the actions of an uncontrollable environment.

Designs in formal verification are often described as a collection of concurrent non-deterministic modules. For instance, the input language to the model checker SMV [119] defines each module as a set of state variables and an expression stating the possible assignments to the variables. Modules in digital circuit description languages such as VHDL and Verilog [16, 15] describe a unit in the circuit at some level of detail by defining the computations mapping signals from input to output wires. Similarly to NADL$^+$, designs in these languages typically describe a closed system where both the behavior of the system and its environment are defined. However, it is not obvious how to use these languages as planning languages since a design is assumed to describe a controlled system.

In DES control theory, systems are often described visually using Petri nets [136] or object oriented description languages such as the Unified Modeling Language (UML) [22]. These representations, however, often grow fast with the size of the system and are, like design representations in formal verification, often describing a controlled system.

8.6 Summary

In this chapter, we have discussed work related to the thesis. The investigation of related work is based on previous work in AI, DES control theory, formal verification, game theory, and automata theory. The main conclusions are

1. Using BDDs for non-deterministic search and for representing non-deterministic plans seems to be the currently most efficient approach to non-deterministic planning for domains with dense non-determinism,

2. State-set branching appears to be the currently most general and most computationally efficient framework for combining classical heuristic search and BDD-based search,

3. Non-deterministic state-set branching is, as far as we know, the first framework for

guiding BDD-based search algorithms that generate non-deterministic plans as defined in this thesis,

4. The fault tolerant planning algorithms introduced in the thesis are to our knowledge the first algorithms to synthesize n-fault tolerant control strategies given a domain description that explicitly represents successful and failure effects of actions.

5. Adversarial planning is, as far as we know, the first work that studies fully implemented and complete symbolic algorithms for synthesizing strategies for winning concurrent reachability games with probability 1 or positive probability. To our knowledge, it also is the first work that provides such algorithms in a format that enables guided search techniques to be applied.

6. NADL$^+$ is to our knowledge the first representation language suitable for planning that both explicitly represents uncontrollable environment actions and failure effects of actions.

Chapter 9

Conclusion

In this chapter, we first briefly summarize the main contributions of the thesis in Section 9.1. Then in Section 9.2, we consider non-deterministic planning as a possible future approach to automated controller synthesis.

9.1 Contributions

The goal of this thesis has been to push the current state-of-the-art of BDD-based non-deterministic planning in two independent directions. The first of these is to develop BDD-based non-deterministic planning algorithms with high performance. To this end, we have developed a general framework called *state-set branching* that seamlessly combines deterministic BDD-based search and classical heuristic search. Our experimental results show that the performance of a state-set branching implementation of the A* algorithm often dominates both blind BDD-based search and the ordinary A* algorithm. In addition, it consistently outperforms the previous BDD-based implementation of A*. We have shown that state-set branching generalizes to non-deterministic planning and have introduced heuristically guided algorithms for weak, strong cyclic, and strong non-deterministic planning. Our experimental results show that extensive performance gains can be obtained with these algorithms compared to the ordinary blind BDD-based search algorithms, both in terms of computational efficiency and the size of the produced plans.

The second direction of work in the thesis is to improve the current solution classes in BDD-based non-deterministic planning. To this end, we have introduced two new frmaeworks called *fault tolerant planning* and *adversarial planning*. Fault tolerant planning extends the non-deterministic domain model with an explicit description of the effect of failing actions. In this way, it is possible to define a new class of non-deterministic plans called *n-fault tolerant plans*. Compared to strong cyclic and strong plans, the advantage of fault tolerant plans is that they do not have to take all possible fault combinations into account. N-fault tolerant plans guarantee goal achievement, but only if no more than n faults occur during execution. The fault tolerant planning algorithms introduced in the thesis are the first to synthesize n-fault tolerant control strategies given a domain description that explicitly represents successful and failure effects of actions.

Adversarial planning extends the non-deterministic domain model with a set of uncontrollable environment actions that causes the outcome of the controllable actions to be non-deterministic. We show that ordinary strong cyclic plans may never reach a goal state if the environment is an informed opponent. To address this problem, we introduce two classes of adversarial plans called weak adversarial plans and strong cyclic adversarial plans. We present two BDD-based algorithms for computing these plans. The algorithms are extensions of the previous weak and strong cyclic planning algorithms and may be defined in a guided version using non-deterministic state-set branching. To our knowledge, adversarial planning is the first work that studies fully implemented and complete symbolic algorithms for synthesizing strategies for winning concurrent reachability games with probability 1 or positive probability.

The thesis demonstrates that BDD-based non-deterministic planning can scale to significant real-world domains such as the Deep Space 1 domain, the SIDMAR steel producing plant, and the Power Supply Restoration domain (PSR). The thesis, however, does not include experimental work on executing these plans in order to control the physical systems they model. There are several issues that must be considered when applying non-deterministic plans.

1. The plans do not provide any probability distribution over the actions to apply in a state. For some domains this may restrict the useability of the plans.

2. Except for adversarial plans, it is assumed that only a single activity takes place in each time step. The activities may represent events in a discrete event system, but it is not natural to use a planning language like NADL$^+$ to encode such domains. In addition, the non-deterministic model does not provide a solution to the timing problem of these events.

3. In order to select actions from a plan, the current state must be fully observable. Often this is not the case. The problem may be addressed by abstracting the domain model to the subset of observable state variables. This, however, may restrict the ability to produce practically useful plans.

Despite these constraints, we believe that there exists important applications where the non-deterministic abstraction or one of the extensions described in the thesis are strong enough to produce practically useful solutions. In addition, we are convinced that the simplicity of non-deterministic abstraction compared to for example Markov decision processes or timed automata may be necessary to scale to the extremely large domains often considered in real-world applications. For these reasons, we recommend a continued focus on this line of research in the future.

9.2 Outlook and Future Directions

An interesting direction of future work is to use non-deterministic BDD-based planning for automated controller synthesis. Non-deterministic plans correspond to discrete, memoryless, and untimed controllers where the task is to force the controlled system into a set of goal states. There is a wide range of high profile application domains for such controllers including automated production, traffic control, robotics, and embedded systems, just to mention a few. A surprising fact is that the current efforts on developing efficient controller synthesis algorithms are very limited. Automated planning has a strong focus on developing efficient data structures and algorithms to make planning systems scale. There has been a significant amount of work on non-deterministic domains and robotics applications. However, automated planning has not traditionally had close ties to the application domains mentioned above. In DES control theory, the situation is the opposite. There has always been a close connection to industrial applications, but the efforts on developing efficient algorithms and data structures for automated DES controller synthesis have been limited. The reason for this may be partially historical since the programming of control switching boards mainly has been considered a technician field [109]. The game tree algorithms [125] and real time reactive planning algorithms [105] developed in AI are probably some of the most scalable approaches to automated discrete control known today. A limitation of these algorithms, however, is that they are incomplete and may drive the system into an unrecoverable state that could have been avoided given a complete search prior to execution. In formal verification there has been extensive work on automated verification of discrete controllers (e.g.,[11]). However, the amount of work on controller synthesis is limited.

Several major questions need to be answered in order to mature BDD-based non-deterministic planning for automated controller synthesis. First of all, a library of industrial benchmark problems needs to be established similar to the benchmark suites used in formal verification. This will clarify the distribution and character of the relevant problems and help to guide a development of specialized algorithms for key problems. Second, an appropriate family of specification languages and domain description languages must be developed. The current approach in DES control theory is to use Petri nets or complete state transition graphs to represent domains and specifications and does not seem to scale to large and combinatorially complex problems. Finally, it needs to be clarified how efficiently controllers represented by BDDs can be mapped to integrated circuits. If circuit representations of controllers tend to be large this strongly limits the applicability of the approach.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[2] L. Alfaro and T. A. Henzinger. Concurrent omega-regular games. In *Proceedings of the 15th Annual Symposium on Logic in Computer Science (LICS)*, pages 141–154, 2000.

[3] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems*, pages 1–20, 1994.

[4] F. Bacchus. AIPS'00 planning competition : The fifth international conference on artificial intelligence planning and scheduling systems. *AI Magazine*, 22(3):47–56, 2001.

[5] F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. In M. Ghallab and A. Milani, editors, *New directions in AI planning*, pages 141–153. ISO Press, 1996.

[6] R. Bahar, E. Frohm, C. Gaona, E Hachtel, A Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *IEEE/ACM International Conference on CAD*, pages 188–191, 1993.

[7] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Trans. on Automatic Control*, 38(7), 1993.

[8] A. Barret and D. S. Weld. Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71–112, 1994.

[9] S. Bart. Department of Computer Science, Cornell University, Ithaca, USA, Personal communication, March 2003.

[10] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[11] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W Yi. UPPAAL - A tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243, 1995.

[12] P. Bertoli, A. Cimatti, and M. Roveri. Conditional planning under partial observability as heuristic-symbolic search in belief space. In *Pre-Proceedings of the 6th European Conference on Planning (ECP-01)*, pages 379–384, 2001.

[13] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in non-deterministic domains under partial observability via symbolic model checking. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 473–478, 2001.

[14] P. Bertoli, A. Cimatti, J. Slanley, and S. Thiébaux. Solving power supply restoration problems with planning via symbolic model checking. In *Proceedings of the 15th European Conference on Artificial Intelligence ECAI'02*, 2002.

[15] J. Bhasker. *A Verilog HDL Primer*. Star Galaxy Publishing, second edition, 1999.

[16] J. Bhasker. *A VHDL Primer*. Prentice Hall, third edition, 1999.

[17] R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Proceedings of the 37th Design Automation Conference (DAC'00)*, pages 29–34. ACM, 2000.

[18] A. Blum and M. L. Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1636–1642, 1995.

[19] A. Bockmayr and D. Dimopoulos. Mixed integer programming models for planning problems. In *Working Notes of the CP-98 Constraint Problem Reformulation Workshop*, 1998.

[20] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proceedings of the 5th European Conference on Planning (ECP-99)*, pages 360–372. Springer, 1999.

[21] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI'97)*, pages 714–719. AAAI Press, 1997.

[22] G. Booch, J. Rumbaugh, and I. Jacobsen. *The Unified Modelling Language User Guide*. Addison Wesley, 1998.

[23] M. H. Bowling, R. M. Jensen, and M. M. Veloso. A formalization of equilibria for multiagent planning. In *18th National Conference on Artificial Intelligence (AAAI-02) workshop on Planning with and for Multiagent Systems*, 2002.

[24] J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems 7 (NIPS)*, 1995.

[25] K. Brace, R. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[26] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 8:677–691, 1986.

[27] J. R. Burch, E. M. Clarke, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[28] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, pages 49–58. North-Holland, 1991.

[29] T. Bylander. Complexity results for serial decomposability. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92)*, pages 729–734, 1992.

[30] J. G. Carbonell. Counterplanning: A strategy-based model of adversary planning in real-world situations. *Artificial Intelligence*, 16(3):257–294, 1981.

[31] J. Chen and R. J. Patton. *Robust Model-Based Fault Diagnosis for Dynamic Systems*. Kluwer Academic Publishers, 1999.

[32] K.-H. Cho and J.-T. Lim. Synthesis of fault tolerant supervisor for automated manufacturing systems: A case study on photolithographic process. *IEEE Trans. on Robotics and Automation*, pages 348–351, 1998.

[33] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for $\mathcal{AR}$. In *Proceedings of the 4th European Conference on Planning (ECP'97)*, pages 130–142. Springer, 1997.

[34] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147(1-2), 2003. Elsevier Science publishers.

[35] A. Cimatti and M. Roveri. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*, 13:305–338, 2000.

[36] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 875–881. AAAI Press, 1998.

[37] A. Cimatti, M. Roveri, and P. Traverso. Strong planning in non-deterministic domains via model checking. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning System (AIPS'98)*, pages 36–43. AAAI Press, 1998.

[38] M. D. Cin. Verifying fault-tolerant behavior of state machines. In *Proceedings of the Second IEEE High-Assurance Systems Engineering Workshop HASE 97*, pages 97–99, 1997.

[39] E. Clarke. Computer Science Department, Carnegie Mellon University, Pittsburgh, USA, Personal communication, May 2003.

[40] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[41] O. Coudert, C. Berthet, and J. Madre. Verification of sequential machines using symbolic execution. *Automatic Verification Methods for Finite State Machines*, pages 365–373, 1989.

[42] M. Daniele, P. Traverso, and M. Y. Vardi. Strong cyclic planning revisited. In *Proceedings of the Fifth European Conference on Planning (ECP'99)*, pages 35–48. Springer-Verlag, 1999.

[43] L. De Alfaro, Henzinger T. A., and O. Kupferman. Concurrent reachability games. In *IEEE Symposium on Foundations of Computer Science*, pages 564–575, 1998.

[44] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Association for Computing Machinery*, 32(3):505–536, 1985.

[45] R. J. Doyle. Determining the loci of anomalies using minimal causal models. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1821–1827, 1995.

[46] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of 2nd Conference on Artificial Intelligence Planning Systems (AIPS'94)*, 1994.

[47] E. H. Durfee. *Coordination of Distributed Problem Solvers*. Kluwer Academic Press, 1988.

[48] S. Edelkamp. Directed symbolic exploration in AI-planning. In *AAAI Spring Symposium on Model-Based Validation of Intelligence*, pages 84–92, 2001.

[49] S. Edelkamp. Symbolic exploration in two-player games: Preliminary results. In *Proceedings of the International Conference on AI Planning and Scheduling (AIPS'02) Workshop on Model Checking*, 2002.

[50] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proceedings of the 6th European Conference on Planning (ECP'99)*, pages 135–147, 1999.

[51] S. Edelkamp and M. Helmert. On the implementation of MIPS. In *Proceedings of AIPS-2000 Workshop on Decision-Theoretic Planning*, pages 18–25, 2000.

[52] S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-spin. In *Proceedings of SPIN-01*, pages 57–79, 2001.

[53] S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *Proceedings of the 22nd Annual German Conference on Advances in Artificial Intelligence (KI-98)*, pages 81–92. Springer, 1998.

[54] S. Edelkamp and F. Reffel. Deterministic state space planning with BDDs. In *Proceedings of the 5th European Conference on Planning (ECP-99)*, pages 381–382, 1999.

[55] E. A. Emerson and J. Srinivasan. Branching time temporal logic. In J. W. Bakker, W. P. Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 123–172. Springer, Berlin, 1989.

[56] A. Fehnker. Scheduling a steel plant with timed automata. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, 1999.

[57] Z. Feng and E. Hansen. Symbolic LAO* search for factored markov decision processes. In *Proceedings of the AIPS-02 Workshop on Planning via Model Checking*, pages 49–53, 2002.

[58] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[59] M. P. Fourman. Propositional planning. In *Proceedings of the AIPS-00 Workshop on Model-Theoretic Approaches to Planning*, pages 10–17, 2000.

[60] M. Fox and D. Long. The PDDL 2.1 home page. `http://www.dur.ac.uk/d.p.long/IPC/pddl.html`, 2002.

[61] G. Gabodi, P. Camurati, L. Lavagno, and S. Quer. Disjunctive partitioning and partial iterative squaring. In *Proceedings of the 34th Design Automation Conference DAC-97*, 1997.

[62] M. Georgeff and A. L. Lansky. Procedural knowledge. *Proceedings of IEEE*, 74(10):1383–1398, 1986.

[63] M. P. Georgeff. Communication and interaction in multiagent planning. In *Proceedings of the 3rd National Conference on Artificial Intelligence (AAAI'83)*, pages 125–129, 1983.

[64] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 905–912, 1998.

[65] M. L. Ginsberg. Universal planning: An (almost) universal bad idea. *AI Magazine*, 10(4):40–44, 1989.

[66] E. Giunchiglia, G. N. Kartha, and Y. Lifschitz. Representing action: Indeterminacy and ramifications. *Artificial Intelligence*, 95:409–438, 1997.

[67] F. Giunchiglia, L. Spalazzi, and P. Traverso. Planning with failure. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*, 1994.

[68] R. Goldman and M. Boddy. Expressive planning and explicit knowledge. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS'96)*, pages 110–117, 1996.

[69] A. Groce and W. Visser. Heuristic model checking for java programs. In *Proceedings of the SPIN Workshop on Model Checking of Software*, pages 242–245, 2002.

[70] P. Haddawy and M. Suwandi. Decision-theoretic refinement planning using inheritance abstraction. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS-92)*, 1994.

[71] K. Z. Haigh and M. M. Veloso. Planning, execution and learning in a robotic agent. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS'98)*, pages 120–127. AAAI Press, 1998.

[72] K. Hammond. *Case-Based Planning: Viewing Planning as a Memory Task*. Academic Press, 1989.

[73] K.J. Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 40:173–228, 1990.

[74] E. Hansen, R. Zhou, and Z. Feng. Symbolic heuristic search using decision diagrams. In *Symposium on Abstraction, Reformulation and Approximation SARA'02*, 2002.

[75] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on SSC*, 100(4), 1968.

[76] J. Hoey, R. St-Aubin, and A. Hu. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*, pages 279–288, 1999.

[77] J. Hoffman and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Submitted, Journal of Artificial Intelligence Research*, 2001.

[78] G. Hoffmann and H. Wong-Toi. Symbolic synthesis of supervisory controllers. In *Proceedings of 1992 American Control Conference*, pages 2789–2793, 1992.

[79] J. Hoffmann. Local search topology in planning benchmarks: An empirical analysis. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 453–458. Morgan Kaufmann, 2001.

[80] S. Holldouble and H.-P. Stör. Solving the entailment problem in the fluent calculus using binary decision diagrams. In *AIPS-2000 Workshop on Model-Theoretic Approaches to Planning*, pages 32–39, 2000.

[81] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[82] H. Iwashita and T. Nakata. Forward model checking techniques oriented to buggy designs. In *IEEE/ACM International Conference on Computer-Aided Design (CAD-97)*, pages 400 –404, 1997.

[83] N. R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75(2):195–240, 1995.

[84] R. M. Jensen. OBDD-based universal planning in multi-agent, non-deterministic domains. Master's thesis, Technical University of Denmark, Department of Automation, 1999. IAU99F02.

[85] R. M Jensen. A comparison study between the CUDD and BuDDy OBDD package applied to AI-planning problems. Technical report, Computer Science Department, Carnegie Mellon University, 2002. CMU-CS-02-173.

[86] R. M. Jensen. Efficient BDD-based search for planning, thesis proposal. `http://www.cs.cmu.edu/~runej`, 2002.

[87] R. M. Jensen. The BDD-based InFoRmed planning and cOntroller Synthesis Tool BIFROST version 0.7. `http://www.cs.cmu.edu/~runej`, 2003.

[88] R. M. Jensen, R. E. Bryant, and M. M. Veloso. An efficient BDD-based A* algorithm. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02) Workshop on Planning via Model Checking*, 2002.

[89] R. M. Jensen, R. E. Bryant, and M. M. Veloso. SetA*: An efficient BDD-based heuristic search algorithm. In *Proceedings of 18th National Conference on Artificial Intelligence (AAAI'02)*, pages 668–673, 2002.

[90] R. M. Jensen, R. E. Bryant, and M. M. Veloso. SetA* applied to channel routing. Technical report, Computer Science Department, Carnegie Mellon University, 2002. CMU-CS-02-172.

[91] R. M. Jensen, R. E. Bryant, and M. M. Veloso. State-set branching: Leveraging OBDDs for heuristic search. *Artificial Intelligence*, To Appear.

[92] R. M. Jensen and M. M. Veloso. OBDD-based deterministic planning using the UMOP planning framework. In *Proceedings of the AIPS-00 Workshop on Model-Theoretic Approaches to Planning*, pages 26–31, 2000.

[93] R. M. Jensen and M. M. Veloso. OBDD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research*, 13:189–226, 2000.

[94] R. M. Jensen, M. M. Veloso, and M. Bowling. Optimistic and strong cyclic adversarial planning. In *Pre-proceedings of the 6th European Conference on Planning (ECP'01)*, pages 265–276, 2001.

[95] R. M. Jensen, M. M. Veloso, and R. E. Bryant. Guided symbolic universal planning. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling ICAPS-03*, pages 123–132, 2003.

[96] R. M. Jensen, M. M. Veloso, and R. E. Bryant. Synthesis of fault tolerant plans for non-deterministic domains. In *Proceedings of ICAPS'03 Workshop on Planning under Uncertainty and Incomplete Information*, pages 64–73, 2003.

[97] M. Jurdzinski, O. Kupferman, and T. A. Henzinger. Trading probability for fairness. In *Proceedings of the International Conference for Computer Science Logic (CSL)*, pages 292–305, 2002.

[98] F. Kabanza, M. Barbeau, and R. St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95:67–113, 1997.

[99] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, volume 2, pages 1194–1201. AAAI Press, 1996.

[100] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, volume 1, pages 318–325. Morgan Kaufmann, 1999.

[101] H. Kautz and J. Walser. State-space planning by integer optimization. In *Proceedings of National Conference on Artificial Intelligence (AAAI'99)*, 1999.

[102] J. Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[103] E. Klein and H. Wehlan. Systematic design of a protective controller in process industries by means of the boolean differential calculus. In *Proceedings of WODES-96*, 1996.

[104] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *Proceedings of the 4th European Conference on Planning (ECP'97)*, Lecture Notes in Artificial Intelligence, pages 273–285. Springer-Verlag, 1997.

[105] S. Koenig and R. G. Simmons. Real-time search in non-deterministic domains. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1660–1667. Morgan Kaufmann, 1995.

[106] R. Korf. Real-time heuristic search: First results. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)'87*, pages 133–138, 1987.

[107] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[108] T. Kreifelts and F. Martial. A negotiation framework for autonomous agents. In *Proceedings of the 2nd European Workshop on Modeling Autonomous Agents and Multi-Agent Worlds*, pages 169–182, 1990.

[109] B. Krogh. Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, USA, Personal communication, Nov 2002.

[110] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76:239–286, 1995.

[111] J. Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B. MIT Press, 1994.

[112] J. Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. Technical Report IT-TR: 1999-028, Institute of Information Technology, Technical University of Denmark, 1999. `http://cs.it.dtu.dk/buddy`.

[113] D. Long. The AIPS-98 planning competition. *AI Magazine*, 21(2):13–34, 2000.

[114] D. Long and M. Fox. Type analysis of planning domain descriptions. In *Proceedings of 17th Workshop of UK Planning and Scheduling SIG*, 1998.

[115] D. Long and M. Fox. The AIPS-02 planning competition. `http://www.dur.ac.uk/d.p.long/competition.html`, 2002.

[116] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'91)*, pages 634–639, 1991.

[117] G. McCalla and B. Ward. Error detection and recovery in a dynamic planning environment. In *Proceedings of the 2nd National Conference on Artificial Intelligence (AAAI'82)*, pages 172–175, 1982.

[118] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - the planning domain definition language. Technical report, Yale Center for Computational Vision and Control, 1998.

[119] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[120] C. Meinel and Stangier C. A new partitioning scheme for improvement of image computation. In *Proceedings ASP-DAC'2001*, pages 97–102, 2001.

[121] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer, 1998.

[122] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[123] I. Moon, J. H. Kukula, K. Ravi, and F Somenzi. To split or to conjoin: The question in image computation. In *Proceedings of the 37th Design Automation Conference*, pages 23–28, 2000.

[124] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[125] A. Newell, J. C. Shaw, and H. A. Simon. Chess playing programs and the problem of complexity. *IBM Journal of Research Development*, 4(2):320–335, 1958.

[126] A. Newell, J. C. Shaw, and H. A. Simon. Report on a general problem solving program for a computer. In *Proceedings of the International Conference on Information Processing*, 1960.

[127] M. J. Osborne and A. Rubinstein. *A course in game theory*. MIT Press, 1994.

[128] C. M. Özveren and A. S. Willsky. Stability and stabilizability of discrete event dynamic systems. *Journal of ACM*, pages 730–752, 1991.

[129] K. M. Passino. Lyapunov stability of a class of discrete event systems. *IEEE Trans. on Automatic Control*, pages 269–279, 1994.

[130] J. Pearl. *Heuristics : Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[131] C. Pecheur and R. Simmons. From livingstone to SMV. In *FAABS*, pages 103–113, 2000.

[132] E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the 1'st International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332. Morgan Kaufmann, 1989.

[133] J. S. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the 3'rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–114. Morgan Kaufmann, 1992.

[134] M. Peot and D. Smith. Conditional nonlinear planning. In *Proceedings of the 1'st International Conference on Artificial Intelligence Planning Systems (AIPS'92)*, pages 189–197. Morgan Kaufmann, 1992.

[135] T. S. Perraju, S. P. Rana, and S. P. Sarkar. Specifying fault tolerance in mission critical systems. In *Proceedings of High-Assurance Systems Engineering Workshop, 1996*, pages 24–31. IEEE, 1997.

[136] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3), 1977.

[137] B. Piergiorgio, B. Bonet, A. Cimatti, E. Giunchiglia, K. Golden, J. Rintanen, and D. E. Smith. The NuPDDL home page. `http://sra.itc.it/tools/mbp/#nupddl`, 2002.

[138] M. Pistore, R. Bettin, and P. Traverso. Symbolic techniques for planning with extended goals in non-deterministic domains. In *Pre-Proceedings of the 6th European Conference on Planning (ECP-01)*, pages 253–264, 2001.

[139] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computations Science (FOCS-77)*, pages 46–57, 1977.

[140] I. Pohl. First results on the effect of error in heuristic search. *Machine Intelligence*, 5:127–140, 1970.

[141] R. Punkunus. Approximation algorithms for STRIPS reachability analysis. Final Report, Research Elective Course, Computer Science Department, Carnegie Mellon University, 2001.

[142] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.

[143] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM Proceedings of the International Workshop on Logic Synthesis*, 1995.

[144] F. Reffel and S. Edelkamp. Error detection with directed symbolic model. In *Proceedings of World Congress on Formal Methods (FM)*, pages 195–211. Springer, 1999.

[145] J. Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research (JAIR)*, 10:323–352, 1999.

[146] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 139–144, 1993.

[147] S. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach*. Prentice-Hall, 1995.

[148] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[149] E. D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI-75)*, pages 206–214, 1975.

[150] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *IEEE Trans. on Automatic Control*, 40(9):1555–1575, 1995.

[151] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Failure diagnosis using discrete-event models. *IEEE Trans. on Control Systems Technology*, 4(2):105–123, 1996.

[152] M. Sampath, R. Sengupta, S. Lafortune, and D. Teneketzis. Active diagnosis of discrete-event systems. *IEEE Trans. on Automatic Control*, 43(7):908–929, 1998.

[153] F. Schmiedle, R. Drechsler, and B. Becker. Exact channel routing using symbolic representation. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS'1999)*, 1999.

[154] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 1039–1046. Morgan Kaufmann, 1987.

[155] R. Senjen and M. De Beler. Hybrid expert systems for monitoring and fault diagnosis. In *Proceedings of the 9th IEEE Conference on Artificial Intelligence Applications*, pages 235–241, 1993.

[156] L. S. Shapley. Stochastic games. *PNAS*, 39:1095–1100, 1953.

[157] S. J. J. Smith and D. S. Nau. Total-order multi-agent task network planning for contract bridge. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI'96)*, 1996.

[158] F. Somenzi. CUDD: Colorado university decision diagram package. `ftp://vlsi.colorado.edu/pub/`, 1996.

[159] R. Su. Decentralized fault diagnosis for discrete-event systems. Master's thesis, Dept. Electl. Engrg., Univ. of Toronto, 2001.

[160] K. Sulimma and K. Wolfgang. An exact algorithm for solving difficult detailed routing problems. In *Proceedings of the 2001 International Symposium on Physical Design*, pages 198–203, 2001.

[161] R. S. Sutton and A. G. Barto. *Reinforcement Learning: an Introduction*. MIT Press, 1998.

[162] S. Thiébaux and M. O. Cordie. Supply restoration in power distribution systems – a benchmark for planning under uncertainty. In *Pre-Proceedings of the 6th European Conference on Planning (ECP-01)*, pages 85–96, 2001.

[163] M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[164] M. M. Veloso. *Planning and Learning by Analogical Reasoning*. Springer-Verlag, 1994.

[165] M. M. Veloso and J. Blythe. Linkability: Examining causal link commitments in partial-order planning. In *Proceedings of the Second International Conference on AI Planning Systems (AIPS'94)*, pages 170–175, 1994.

[166] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *IEEE International Conference on Automated Software Engineering (ASE)*, 2000.

[167] D. H. D. Warren. Generating conditional plans and programs. In *Proceedings of the AISB Summer Conference*, pages 344–354, 1976.

[168] I. Wegener. *Branching Programs and Binary Decision Diagrams*. Society for Industrial and Applied Mathematics (SIAM), 2000.

[169] D. S. Weld, C. R. Anderson, and D. E. Smith. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, 1998.

[170] D. Wilkins. Using patterns and plans in chess. *Artificial Intelligence*, 14:165–203, 1980.

[171] D. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufman, 1988.

[172] D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical Artificial Intelligence*, 6:197–227, 1994.

[173] B. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, 1996.

[174] B. C. Williams, M. Ingham, S. H. Chung, and P. H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. In *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, volume 9, pages 212–237, 2003.

[175] H. Xu. Structured routing with boolean satisfiability. CMU ECE Course Paper, 2001.

[176] N. XuanLong and S. Kambhampati. Extracting effective and admissible state space heuristics from the planning graph. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI'00)*, pages 798–805, 2000.

[177] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of BDD-based model checking. In *Formal Methods in Computer-Aided Design FMCAD'98*, pages 255–289, 1998.

[178] B. Yang, R. Simmons, R. E. Bryant, and R. O. O'Hallaron. Optimizing symbolic model checking for constraint-rich models. In *Proceedings of Computer-Aided Verification (CAV'99)*, pages 328–340, 1999.

[179] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Proceedings of the 35th Design Automation Conference (DAC'98)*, pages 599–604. ACM, 1998.

[180] J. Yuan, J. Shen, J. Abraham, and A. Aziz. Formal and informal verification. In *Conference on Computer Aided Verification (CAV'97)*, pages 376–387, 1997.

[181] G. Zlotkin and J. S. Rosenschein. Incomplete information and deception in multi-agent negotiation. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI'85)*, pages 225–231, 1985.

Appendix A

BIFROST

This appendix contains a description of the Bdd-based InFoRmed planning and cOntroller Synthesis Tool (BIFROST). Section A.1 is a user guide to BIFROST version 0.7. Section A.2 describes the syntax and semantics of NADL$^+$ used as input language to BIFROST. Finally, Section A.3 describes the experimental setting used for the experiments described in the thesis.

A.1 User Guide to BIFROST 0.7

BIFROST version 0.7 is a software package for BDD-based deterministic and non-deterministic planning and heuristic search. The program is written in C++/STL for the GNU GCC compiler running on a Redhat Linux 7.1 PC. The software is open source and may be used for scientific and teaching purposes. BIFROST uses the BuDDy 2.0 BDD-package [112].[1]

A.1.1 Usage

Follow the instructions on the BIFROST web site [87] to download and install the program. BIFROST is a regular UNIX command

```
bifrost -d domainFile [-iaxyvponcrthufeg].
```

The options of BIFROST are shown by executing `bifrost -h`. The input to BIFROST is a planning problem written either in the STRIPS part of PDDL [118] or NADL$^+$ described

[1]Comparison experiments with the CUDD package [158] has not shown a significant performance difference [85].

in Section A.2. Option `-i` *type* defines the input type and must be set. The possible values for *type* are `PDDL` and `NADL`. If the input is PDDL, two input files must be given. The first is the PDDL domain description which is set by option `-d` *domain file name*. The second is the PDDL problem description which is set by option `-p` *problem file name*. If the input is NADL^+, only a single NADL^+ input file is given containing both a domain description and a problem description. The name of the NADL^+ file is set with option `-d` *domain file name*. The verbosity level is set by option `-v` *num* where *num* is a non-negative number. The higher the value of *num*, the more information BIFROST dumps to the screen.

The memory parameters of the BuDDy package are adjusted with options `-n` *num* and `-c` *num*. The `-n` option sets the number of BDD-nodes allocated to represent the shared BDD, while the `-c` option sets the number of BDD nodes allocated to represent the BDDs in the operator caches used to implement dynamic programming. For medium sized problems good values for n and c are around 1M and 400K, respectively. The Buddy package can also be initialized to use dynamic variable reordering with option `-r` *type*. The possible values of *type* are `Off` (no dynamic variable reordering) and `Win2ite` (sliding window reordering).[2]

The search algorithm used by BIFROST is set by option `-a` *type*. The possible values of *type* are shown in Table A.1. All BDD-based algorithms implemented in BIFROST rely on a disjunctive partitioning of the transition relation. The threshold for merging partitions is set by option `-t` *num*. The search timeout bound is set by option `-l` *num* where *num* is the timeout bound in seconds. For bidirectional, forward, and backward deterministic search, frontier set simplification based on [41] can be activated by option `-f`. For the weighted A* algorithms, f is given by $f = x * g + y * h$ where x and y are in the range $[0; 1]$ and are set by options `-x` *num* and `-y` *num*. For PDDL problems, the heuristic function is given by option `-g` *type* where the possible values of *type* are `MinHamming`, which is the minimum Hamming distance, and `HSPr` which is the HSPr heuristic described in [20]. For any of the state-set branching algorithms, option `-u` *num* sets the upper bound of the size of merged BDDs in the search queue.

BIFROST can write two different output files. The first is the solution file. Its name is set by option `-o` *solution file name*. For deterministic problems, it is a text file with a solution given as a sequence of actions. For non-deterministic problems, it is a BDD file representing the produced non-deterministic plan. The second possible output file is for conducting experiments with BIFROST. The name of the experiment file is set by option `-e` *experiment file name*. The experiment file is a text file with data about the search including time to allocate memory, analyse the domain, build the transition relation, and search. In addition, it contains information about the size of the solution, the average size

[2] See the BuDDy 2.0 user manual for a detailed description.

Deterministic Search Algorithms		
`Bidir`	:	BDD-based breadth-first bidirectional search.
`Forward`	:	BDD-based breadth-first forward search.
`Backward`	:	BDD-based breadth-first backward search.
`ghSetAstar`	:	GHSETA* in a weighted version ($f = x * g + y * h$).
`fSetAstar`	:	FSETA* in a weighted version ($f = x * g + y * h$).
`Astar`	:	Ordinary weighted A* with explicit state representation and cycle detection. The input must be in PDDL format.
`BDDAstar`	:	BDDA*.
`iBDDAstar`	:	Improved BDDA*.
Non-Deterministic Search Algorithms		
`Weak`	:	WEAK.
`WeakH`	:	GUIDEDWEAK.
`WeakAdv`	:	WEAKADVERSARIAL. The input must be in NADL$^+$ format.
`StrongCyclic`	:	STRONGCYCLIC.
`StrongCyclicH`	:	GUIDEDSTRONGCYCLIC.
`StrongCyclicAdv`	:	STRONGCYCLICADVERSARIAL. The input must be in NADL$^+$ format.
`Strong`	:	STRONG.
`StrongH`	:	GUIDEDSTRONG.
`FaultTolerant`	:	1-FTP. The input must be in NADL$^+$ format.
`GuidedFaultTolerant`	:	1-GFTP. The input must be in NADL$^+$ format.

Table A.1: BIFROST search algorithms.

of the BDDs representing the search frontier, and the number of iterations of the algorithm. Finally, it summarizes the parameters of the BDD package and the name of the input file. If an experiment file already exists with the same name, BIFROST appends its result to the file. Otherwise, it creates the file and adds the first row of results.

A.1.2 Examples

```
bifrost -i NADL -d 5line.nadl -a WeakH -t 5000 -e WeakH.dat
-n 15000000 -c 500000 -v 1 -l 1000
```

Initializes the Buddy package with 15M BDD nodes and a cache of 500K BDD nodes. BIFROST then builds a disjunctive partitioning of the NADL$^+$ problem described in the file `5line.nadl` with a merging threshold of 5000 BDD nodes. The GUIDEDWEAK non-deterministic planning algorithm is used to find a solution. The timeout bound is set to 1000 seconds and the debug verbosity level is 1. The experimental results are written to the file `WeakH.dat`.

```
bifrost -i PDDL -d domain.pddl -a BDDAstar -v 1 -e BDDAstar
.dat -p single01.pddl -t 4000 -n 8000000 -c 400000 -g HSPr
```

Initializes the Buddy package with 8M BDD nodes and a cache of 400K BDD nodes. BIFROST then builds a disjunctive partitioning of the PDDL domain and problem described in the files `domain.pddl` and `single01.pddl` with a merging threshold of 4000 BDD nodes. The BDDA* algorithm is used to find a solution using the HSPr heuristic. The timeout bound is set to the default 500 seconds and the debug verbosity level is 1. The experimental results are written to the file `BDDAstar.dat`.

```
bifrost -i NADL -d D4V4M15.nadl -g MinHamming -l 500 -u 200
-n 8000000 -c 700000 -x 1.0 -y 1.0 -t 5000 -e ghSetAstar.exp
-a ghSetAstar
```

Initializes the Buddy package with 8M BDD nodes and a cache of 700K BDD nodes. BIFROST then builds a disjunctive partitioning of the NADL$^+$ problem described in the file `D4V4M15.nadl` with a merging threshold of 5000 BDD nodes. The GHSETA* algorithm with $f = 1.0*g+1.0*h$ is used to find a solution using the min Hamming distance heuristic. BDD nodes with a size below 200 are merged in the search queue of GHSETA*. The timeout bound is set to 500 seconds and the debug verbosity level is 1. The experimental results are written to the file `ghSetAstar`.

A.2 NADL$^+$

NADL was developed as a part of the UMOP project [84, 93, 92, 94]. However, despite providing a very general framework for modeling non-deterministic planning problems, NADL does not allow additional information about transition costs, heuristic estimates, and failure effects of actions. NADL$^+$ adds these features to the language. There are are three main differences between the two languages

1. NADL$^+$ has three new optional action description components **dg**, **dh**, and **err**. In addition, it uses the entry **heu** to define the value of the heuristic estimate in the initial state and the goal states,

2. An action description may concist of descriptions of several *transition groups*,

3. NADL$^+$ assumes that the system and environment are described by as set of actions instead of a set of agents.

The action component **dg:** int associates a transition cost or weight with the action. The component **dh:** int describes the change of a heuristic estimate associated with each transition represented by the transition group. The change is always given in forward direction even if the heuristic guides a backward search. Finally, **err:** $formula$ defines a set of next states reached by the action given that its execution fails.

An NADL$^+$ problem description consists of: a set of *state variables*, a set of *system* and *environment actions*, and an *initial* and *goal condition*. The set of state variable assignments defines the state space of the domain. The set of system actions must be non-empty while the set of environment actions may be empty if no active environment exists. System and environment actions are assumed to be *synchronous*. At each step, exactly a single system and environment action is performed. The resulting action is called a *joint action*. Only the system actions are controllable. An action has three main parts: a set of *modified* state variables, a *precondition* formula, and an *effect* formula. The set of modified variables are the state variables which may have their value changed by the action. In order for an action to be applicable, the precondition formula must be satisfied in the current state. The effect of the action is defined by the effect formula. The value of state variables not modified by a joint action is unchanged. The initial and goal condition are formulas that must be satisfied in the initial state and the goal states, respectively.

Example A.1 An NADL$^+$ planning problem is shown in Figure A.1. The problem has two state variables pos and $power$. The position is a natural number that can be represented by three Boolean variables. This gives pos the domain $\{0, 1, 2, 3, 4, 5, 6, 7\}$. The power is a proposition and is represented by a single Boolean state variable. The system is a robot moving between the eight positions. It has two actions *Right* and *Left*. The cost of both actions is 1. The heuristic is for guiding a backward search from the goal states to the initial state. It therefore estimates the distance to the initial state. This estimate is simply the value of the position. Thus, a successful Left action changes the heuristic estimate with -1, while a successful Right action changes it with $+1$. The effect of the Right action, depends on the $power$ variable. If the power is $true$ then the position is increased, otherwise nothing happens. For this reason, the transitions of the Right action are partitioned into two transition groups where the first describes the successful outcome of the action where **dh:** is 1, and the second describes the unsuccessful outcome of the action where **dh:** is 0. The Left action is assumed to succeed independent of the value of $power$. It can therefore be described by a single transition group. The environment controls the power with two actions *On* and *Off*. Since the system and environment must apply exactly one action at each step, there are four joint actions *Left-On*, *Left-Off*, *Right-On*, and *Right-Off*. Initially, the power is on and the robot is at position 0. The goal is to reach position 7. The value of the heuristic estimate must be given for the goal states in order to use a branching partitioning to propagate the value of the heuristic estimate to other states. This is done by adding the entry **heu:** 7 to the goal condition. $\Diamond$

Syntax of NADL$^+$

Below is the BNF syntax of NADL$^+$. The syntax of formulas is given separately.

$$
\begin{array}{lcl}
\langle NADL^+\rangle & ::= & \texttt{variables}\ \langle VarDecl\rangle\ \{\langle VarDecl\rangle\} \\
 & & \texttt{system}\ \langle ActionDecl\rangle\ \{\langle ActionDecl\rangle\} \\
 & & \texttt{environment}\ \{\langle ActionDecl\rangle\} \\
 & & \texttt{initially}\ \langle Formula\rangle\ [\texttt{heu} : \langle Number\rangle] \\
 & & \texttt{goal}\ \langle Formula\rangle\ [\texttt{heu} : \langle Number\rangle] \\
 & & \\
\langle VarDecl\rangle & ::= & \langle VarType\rangle\ \langle IdLst\rangle \\
 & & \\
\langle VarType\rangle & ::= & \texttt{bool} \\
 & | & \texttt{nat(}\ \langle Number\rangle\ \texttt{)} \\
 & & \\
\langle IdLst\rangle & ::= & \epsilon \\
 & | & \langle Id\rangle \\
 & | & \langle Id\rangle\ \{,\langle Id\rangle\} \\
 & & \\
\langle ActionDecl\rangle & ::= & \langle Id\rangle\ \langle TranDecl\rangle\ \{\langle TranDecl\rangle\} \\
 & & \\
\langle TranDecl\rangle & ::= & [\texttt{dg} : \langle Number\rangle] \\
 & & [\texttt{dh} : \langle Number\rangle] \\
 & & \ \texttt{mod} : \langle IdLst\rangle \\
 & & \ \texttt{pre} : \langle Formula\rangle \\
 & & \ \texttt{eff} : \langle Formula\rangle \\
 & & [\texttt{err} : \langle Formula\rangle]
\end{array}
$$

variables
  nat(3) pos
  bool $power$
system
    Right
      dg: 1
      dh: 1
      mod: pos
      pre: $pos < 7 \wedge power$
      eff: $pos' = pos + 1$

      dg: 1
      dh: 0
      mod: pos
      pre: $pos < 7 \wedge \neg power$
      eff: $pos' = pos$
    Left
      dg: 1
      dh: -1
      mod: pos
      pre: $pos > 0$
      eff: $pos' = pos - 1$
environment
    On
      mod: $power$
      pre: $\neg power$
      eff: $power'$
    Off
      mod: $power$
      pre: $power$
      eff: $\neg power'$
initially
  $pos = 0 \wedge power$
goal
  $pos = 7$
  heu: 7

Figure A.1: An NADL$^+$ planning problem.

An identifier is a sequence of numbers, letters and the character "_" that does not begin with a number. The syntax of formulas is given below. The $- >$ operator is an *if-then-else* operator. The relation operator $<>$ denotes *not equal to*. The Boolean operators $=>$ and $<=>$ denote logical implication and bi-implication, respectively. The other operators have their usual semantics.

$$
\begin{array}{lcl}
\langle \mathit{Formula} \rangle & ::= & \langle \mathit{Formula} \rangle - > \langle \mathit{Formula} \rangle \text{ , } \langle \mathit{Formula} \rangle \\
 & | & \langle \mathit{Formula} \rangle \ \langle \mathit{BoolOp} \rangle \ \langle \mathit{Formula} \rangle \\
 & | & \langle \mathit{NumExp} \rangle \ \langle \mathit{RelOp} \rangle \ \langle \mathit{NumExp} \rangle \\
 & | & \sim \langle \mathit{Formula} \rangle \\
 & | & (\ \langle \mathit{Formula} \rangle \ ) \\
 & | & \texttt{true} \\
 & | & \texttt{false} \\
 & | & \langle \mathit{Id} \rangle \\
 & & \\
\langle \mathit{BoolOp} \rangle & ::= & => \mid <=> \mid /\backslash \mid \backslash/ \\
 & & \\
\langle \mathit{RelOp} \rangle & ::= & = \mid <> \mid > \mid < \\
 & & \\
\langle \mathit{NumExp} \rangle & ::= & \langle \mathit{Id} \rangle \\
 & | & \langle \mathit{Number} \rangle \\
 & | & \langle \mathit{Number} \rangle \ \langle \mathit{NumOp} \rangle \ \langle \mathit{Number} \rangle \\
\langle \mathit{NumOp} \rangle & ::= & + \mid -
\end{array}
$$

A.3 Experimental Setting

All experiments presented in this thesis have been carried out with BIFROST version 0.7. However, specialized search engines have been implemented for the channel routing experiments and experiments with the ordinary A* algorithm for other heuristics than HSPr. All experiments have been executed on a Redhat Linux 7.1 PC with kernel 2.4.16, 500 MHz Pentium III CPU, 512 KB L2 cache and 512 MB RAM.

PERL scripts are used to execute series of experiments. The results of these experiments are logged in BIFROST experimental files. The experimental files of a particular domain are collected in a master file describing the complete setup of the experiments for future reference and reproduction. The master file also contains a description of the purpose of the experiments and important observations. The main parameters of an experiment, in addition to the problem and the search algorithm, are

1. The n and c parameters of the BuDDy BDD-package, where n is the number of BDD-nodes allocated to represent the shared BDD, and c the number of BDD nodes allocated to represent BDDs in the operator caches used to implement dynamic programming,

2. The threshold t for merging partitions in the disjunctive partitioning given in number of BDDs nodes used to represent the partitions,

3. The upper bound u of the size of merged BDDs in the search queue of the state-set branching algorithms.

Time is measured in seconds. The size of a BDD is equal to the number of nodes in the BDD graph.

Appendix B

Proofs

This appendix contains soundness, completeness, and optimality proofs. The proofs for the WEAK, STRONGCYCLIC, and STRONG algorithms are partially based on previous proofs in [34].

B.1 Notation

Most algorithms consist of an initialization of a set of variables and a main loop that assigns these variables and possibly a set of variables local to the loop. As an example consider the NDP algorithm introduced in Section 3.2.2 and shown below in Figure B.1. The variables

```
function NDP(s_0, G)
P ← ∅; C ← G
while s_0 ∉ C
   P_c ← PRECOMP(C)
   if P_c = ∅ then return "no solution exists"
   else
      P ← P ∪ P_c
      C ← C ∪ STATES(P_c)
return P
```

Figure B.1: The NDP algorithm introduced in Section 3.2.2.

of NDP that are initialized outside the loop and assigned to inside the loop are P and C, while P_c is a local variable of the loop. We will use the following naming conventions for statements about these variables. For a variable V, V_i denotes the value of the variable after i iterations of the loop. Thus, for NDP, C_i, P_i, and P_{c_i} denote the value of C_i, P_i, and P_{c_i} after i executions of the code in line 2 to 7. If V is assigned to several times in an iteration of the loop, V_i refers to its value after the last assignment. If V is initialized before the loop then V_0 denotes its initial value before the first iteration of the loop. If V is a local variable of the loop then V_0 is undefined. Thus, for NDP, P_{c_0} is undefined. V_i is said to exist if the loop iterates at least i times.

B.2 Additional Definitions

This section contains additional definitions used in the proofs.

Definition B.1 (WD) $WD_k(C) \equiv \{s \,:\, \text{WDIST}(s, C) = k\}$.

Definition B.2 (SD) $SD_k(C) \equiv \{s \,:\, \text{SDIST}(s, C) = k\}$.

Definition B.3 (FixedPoint) FIXEDPOINT(C) *is a set of SAs defined by the algorithm below.*

function FIXEDPOINT(C)
$F \leftarrow \emptyset$
repeat
  $F_{old} \leftarrow F$
  $F \leftarrow \text{PREIMGSA}(\text{STATES}(F) \cup C) \setminus C \times Act$
until $F = F_{old}$
return F

Definition B.4 (Adversarial DAG) *An adversarial DAG $AD(C)$ of a set of states C is a graph where the vertices are states and the edges are system actions of a non-deterministic adversarial planning domain. Each state q in an $AD(C)$ is associated with a level $l(q)$. An $AD(C)$ is defined inductively as follows*

- *$c \in C$ are terminal states of $AD(C)$ with $l(c) = 0$,*
- *if $q'_1, \cdots, q'_n$ are states in $AD(C)$ and there for each applicable environment action $Act_e(q) = \{e_1, \cdots, e_n\}$ of a state q exists a counter system action $s_1, \cdots, s_n$ in $\text{APP}_s(q)$ such that $q \xrightarrow{s_i, e_i} q'_i$ for $1 \leq i \leq n$, then q is an internal state of $AD(C)$ with outgoing edges $s_1, \cdots, s_n$ to $q'_1, \cdots, q'_n$. The level of q is $l(q) = \max_{i=1}^{n} l(q'_i) + 1$.*

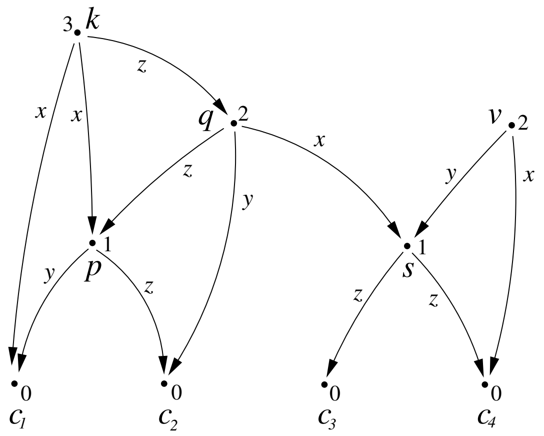


Figure B.2: An adversarial DAG of a set of states $\{c_1, c_2, c_3, c_4\}$. The number shown next to a state is its level.

Notice that an $AD(C)$ is a Directed Acyclic Graph (DAG) since no edge of a state at level l leads to a state at level $l' \geq l$. For a state q in $AD(C)$, let q^{SA} denote the SSAs formed by pairing q with its outgoing edges. Similarly, let $AD^{SA}(C)$ denote the union of the SSAs of each state in $AD(C)$.

Example B.1 Figure B.2 shows an adversarial DAG $AD(C)$ of a set of states $C = \{c_1, c_2, c_3, c_4\}$. It is assumed that the set of system actions is $Act_s = \{x, y, z\}$. We have $AD^{SA}(C)$ $= \{\langle k, x\rangle, \langle k, y\rangle, \langle k, z\rangle, \langle q, z\rangle, \langle q, y\rangle, \langle q, x\rangle, \langle v, y\rangle, \langle v, x\rangle, \langle p, y\rangle, \langle p, z\rangle, \langle s, z\rangle, \langle s, x\rangle\}$. $\diamondsuit$

Definition B.5 (Weak Marking) *A weak marking $AD_w(q, C)$ of an adversarial DAG $AD(C)$ is a subset of $AD(C)$ defined by marking states and edges in $AD(C)$ reachable from q.*

The weak marking $AD_w(q, C)$ is undefined if q is not a state of $AD(C)$.

Example B.2 Figure B.3 shows the weak marking of the adversarial DAG of Example B.1. $\diamondsuit$

Definition B.6 (Strong Cyclic Marking) *A strong cyclic marking $AD_{sc}(q, C)$ of an adversarial DAG $AD(C)$ is a subset of $AD(C)$ defined by, recursively from q, marking each outgoing edge and each state that is reachable by any joint action made up by an applicable environment action and a system counter action.*

The strong cyclic marking $AD_{sc}(q, C)$ is undefined if a state not in $AD(C)$ needs to be marked.

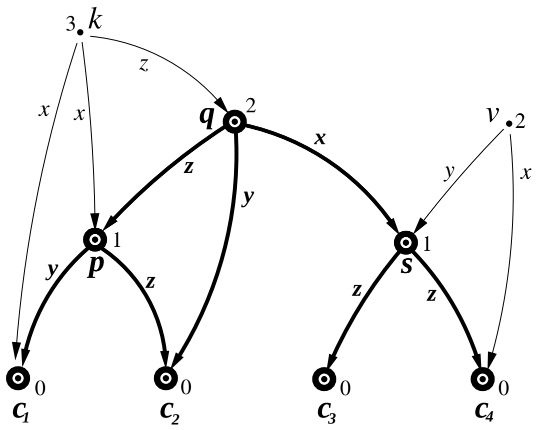


Figure B.3: The weak marking from state q of the adversarial DAG shown in Figure B.2. States and edges in the marking are emphasized.

Example B.3 Figure B.4 shows the strong cyclic marking from q of the adversarial DAG of Example B.1. ◊

B.3 NDP

Let Q_i be the set of states for which a solution is found in iteration i of the while loop of NDP. That is, $Q_0 = G$ and $Q_i = \text{STATES}(P_{c_i})$ for $i > 0$.

Lemma B.1 $C_i = \bigcup_{j=0}^{i} Q_j$.

Proof. This follows directly from $C_0 = G$, $Q_0 = G$, and $C_i = C_{i-1} \cup Q_i$ for $i > 0$. □

Lemma B.2 $Q_i \cap Q_j = \emptyset$ *for* $i \neq j$.

Proof. Assume without loss of generality that $i > j$. Then by Lemma B.1 $C_{i-1} \supseteq Q_j$. By the definition of valid precomponents, we have $C_{i-1} \cap \text{PRECOMP}(C_{i-1}) = \emptyset$, which gives $Q_i \cap Q_j = \emptyset$. □

Lemma B.3 *if* C_i *exists then* $C_i \supset C_{i-1}$.

Proof. If C_i is computed then $C_i = C_{i-1} \cup \text{STATES}(P_{c_i})$ and $\text{STATES}(P_{c_i}) \neq \emptyset$. We have $P_{c_i} = \text{PRECOMP}(C_{i-1})$. By the definition of valid precomponents, we have $\text{STATES}(P_{c_i}) \cap C_{i-1} = \emptyset$. Thus, $C_i \supset C_{i-1}$. □

Theorem B.1 (Termination) *NDP terminates.*

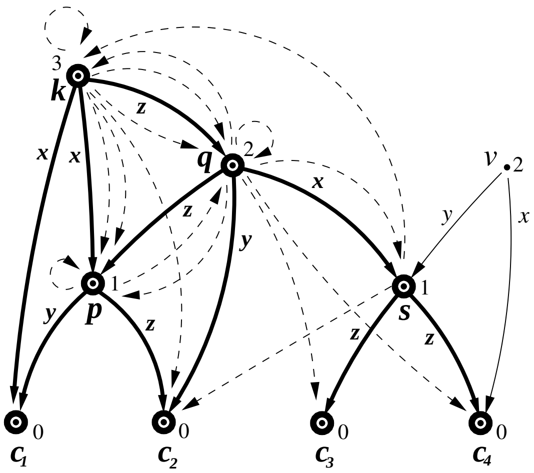


Figure B.4: The strong cyclic marking from state q of the adversarial DAG shown in Figure B.2. States and edges in the marking are emphasized. Dashed edges are not a part of the adversarial DAG but are only used for marking. These edges denote joint actions where the environment action is paired with another system action than its counter action. Consider state q, since there are three edges from this state, we have $|\text{APP}_e(q)| \geq 3$. In the figure, we assume that $|\text{APP}_e(q)| = 3$. This gives a total of 9 joint actions used for marking. Only 3 of these are environment actions paired with their counter system action.

Proof. By the definition of valid precomponents, the function PRECOMP called by NDP terminates. By Lemma B.3, we have that $C_i \supset C_{i-1}$ after completion of iteration i. However, since the state space is finite, the number of iterations then also must be finite. □

B.4 Strong

Lemma B.4 PRECOMPS *is a valid precomponent function.*

Proof. Follows directly from the definition of PRECOMPS and PREIMGSA. □

Lemma B.5 *If* $\pi = \text{PRECOMPS}(C)$ *then* $\mathcal{M}(\pi), \text{STATES}(\pi) \models \texttt{AF}\, C$.

Proof. By definition of $\text{PRECOMP}(C)$, $\pi = \{\langle s, a\rangle \ :\ s \in C \wedge \text{NEXT}(s,a) \cap C \neq \emptyset \wedge \text{NEXT}(s,a) \cap \overline{C} = \emptyset\}$. Thus, for any $s \in \text{STATES}(\pi)$, we have for each s' where $\langle s, s'\rangle \in R$ of $\mathcal{M}(\pi)$ that $s' \in C$, which implies $\mathcal{M}(\pi), s \models \texttt{AF}\, C$. □

Lemma B.6 *For* $\text{STRONG}(s_0, G)$*, we have* $\mathcal{M}(P_i), C_i \models \texttt{AF}\, G$.

Proof. By induction on i.
Case $i = 0$. We have $\mathcal{M}(\emptyset), G \models \mathtt{AF}\, G$, $C_0 = G$, and $P_0 = \emptyset$. Thus, $\mathcal{M}(P_0), C_0 \models \mathtt{AF}\, G$.
Case $i > 0$ The induction hypothesis is $\mathcal{M}(P_{i-1}), C_{i-1} \models \mathtt{AF}\, G$. By Lemma B.5 and $P_{c_i} = \text{PRECOMPS}(C_{i-1})$, we get $\mathcal{M}(P_{c_i}), \text{STATES}(\mathcal{P}_{c_i}) \models \mathtt{AF}\, C_{i-1}$. Combined with the induction hypothesis, we get $\mathcal{M}(P_{c_i} \cup P_{i-1}), \text{STATES}(P_{c_i}) \cup C_{i-1} \models \mathtt{AF}\, G$ which is equal to $\mathcal{M}(P_i), C_i \models \mathtt{AF}\, G$. □

Theorem B.2 (Soundness) STRONG *is sound.*

Proof. If $\text{STRONG}(s_0, G)$ returns a solution π after iteration i then $\pi = P_i$ and $s_0 \in C_i$. Thus, by Lemma B.6, $\mathcal{M}(\pi), s_0 \models \mathtt{AF}\, G$. □

Lemma B.7 *For* $\text{STRONG}(s_0, G)$*, we have* $s \in Q_i \Leftrightarrow \text{SDIST}(s, G) = i$.

Proof. By induction on i.
Case $i = 0$. By definition of Q_0 and SDIST, we have $s \in Q_0 \Leftrightarrow s \in G \Leftrightarrow \text{SDIST}(s, G) = 0$.
Case $i > 0$. The induction hypothesis is $s \in Q_j \Leftrightarrow \text{SDIST}(s, G) = j$ for $j < i$.
"⇒": Assume $s \in Q_i$. By definition of Q_i and PRECOMPS, we get

$$s \in \text{STATES}(\{\langle s', a'\rangle \, : \, s' \notin C_{i-1} \wedge \text{NEXT}(s', a') \cap C_{i-1} \neq \emptyset \wedge \text{NEXT}(s', a') \cap \overline{C}_{i-1} = \emptyset\}).$$

This implies that there exists a set of SAs π such that $\mathcal{M}(\pi), s \models \mathtt{AF}\, C_{i-1}$. Combining this with the induction hypothesis, we get $\text{SDIST}(s) \leq i$. However, if $\text{SDIST}(s) = k < i$ then $s \in Q_k$ according to the induction hypothesis. This is in conflict with the assumption that $s \in Q_i$. Thus, $\text{SDIST}(s, G) = i$.
"⇐": Assume $\text{SDIST}(s, G) = i$. By definition of SDIST, there exists an action a such that $\forall s' \in \text{NEXT}(s, a) \,.\, \text{SDIST}(s', G) \leq i - 1$. Thus, by the induction hypothesis and Lemma B.1, $\text{NEXT}(s, a) \subseteq C_{i-1}$. It must be the case that $s \notin C_{i-1}$ since $s \in C_{i-1}$ contradicts the induction hypothesis. But then by definition of PRECOMPS, $s \in \text{STATES}(\text{PRECOMPS}(C_{i-1}))$ which by definition of Q_i gives $s \in Q_i$. □

Theorem B.3 (Completeness) STRONG *is complete.*

Proof. If a valid solution exists then $\text{SDIST}(s_0, G) \neq \infty$. Assume $\text{SDIST}(s_0, G) = k$. then by definition of SDIST there exists $k+1$ states $q_0, \cdots, q_k$ such that $\text{SDIST}(q_i, G) = i$, $q_k = s_0$, and $q_0 \in G$. Thus, by Lemma B.7, $Q_i \neq \emptyset$ for $i \leq k$ and $s_0 \in Q_k$. Since $Q_i = \text{STATES}(P_{c_i})$, we have $P_{c_i} \neq \emptyset$ for $i \leq k$. Thus, $\text{STRONG}(s_0, G)$ will not terminate with failure in the first k iterations. Also it will not terminate with success in the first $k-1$ iterations since by Lemma B.2, $s_0 \notin Q_i$ for $i < k$. However, since $s_0 \in Q_k$, it will terminate with success in iteration k. □

Lemma B.8 *For* $\text{STRONG}(s_0, G)$*, we have* $\forall s \in Q_i \,.\, \text{MAX}(s, G, P_i) = i$.

Proof. By induction on i.
Case $i = 0$: Let $s \in Q_0$. By definition of Q_0, $s \in G$, but then by definition of MAX, $\text{MAX}(s, G, \emptyset) = \text{MAX}(s, G, P_0) = 0$
Case $i > 0$: The induction hypothesis is $\forall s \in Q_j \,.\, \text{MAX}(s, G, P_j) = j$ for $j < i$. Let $s \in Q_i$. Then by definition of Q_i and $\text{PRECOMPS}(C_{i-1})$, $s \in \text{STATES}(\{\langle s', a' \rangle : s' \notin C_{i-1} \wedge \text{NEXT}(s', a') \cap C_{i-1} \neq \emptyset \wedge \text{NEXT}(s', a') \cap \overline{C}_{i-1} = \emptyset\})$. Thus, by definition of PRECOMPS, $\forall \langle s', a' \rangle \in P_{c_i} \,.\, \text{NEXT}(s', a') \subseteq C_{i-1} \wedge \text{NEXT}(s', a') \cap C_{i-1} \neq \emptyset$. We have $P_i = P_{i-1} \cup P_{c_i}$ and $s \in \text{STATES}(P_{c_i})$. Further, it follows from the definition of C_i and Lemma B.1 and Lemma B.2 that $P_{i-1} \cap P_{c_i} = \emptyset$. Thus, by the induction hypothesis and by definition of MAX, $\text{MAX}(s, G, P_i) \leq i$. However, $\text{MAX}(s, G, P_i) < i$ implies $\text{SDIST}(s, G) < i$ which contradicts Lemma B.7, since we assume $s \in Q_i$. Thus, $\text{MAX}(s, G, P_i) = i$. □

Theorem B.4 (Optimality) *If* π *is a solution returned by* $\text{STRONG}(s_0, G)$ *then*

$$\text{MAX}(s_0, G, \pi) = \text{SDIST}(s_0, G).$$

Proof. Combining Lemma B.7 and Lemma B.8 gives

$$\forall s \in Q_i \,.\, \text{MAX}(s, G, P_i) = \text{SDIST}(s, G)$$

which implies the result. □

B.5 Weak

Lemma B.9 PRECOMPW *is a valid precomponent function.*

Proof. This follows directly from the definition of PRECOMPW and PREIMGSA. □

Lemma B.10 *If* $\pi = \text{PRECOMPW}(C)$ *then* $\mathcal{M}(\pi), \text{STATES}(\pi) \models \text{EF}\, C$.

Proof. By definition of $\text{PRECOMPW}(C)$, we have $\pi = \{\langle s, a \rangle : s \notin C \wedge \text{NEXT}(s, a) \cap C \neq \emptyset\}$. Thus, for any $s \in \text{STATES}(\pi)$, we have $\exists s' \in C \,.\, \langle s, s' \rangle \in R$ of $\mathcal{M}(\pi)$. This implies $\mathcal{M}(\pi), s \models \text{EF}\, C$. Thus, $\mathcal{M}(\pi), \text{STATES}(\pi) \models \text{EF}\, C$. □

Lemma B.11 *For* $\text{WEAK}(s_0, G)$*, we have* $\mathcal{M}(P_i), C_i \models \text{EF}\, G$.

Proof. By induction on i.
Case $i = 0$. We have $\mathcal{M}(\emptyset), G \models \mathtt{EF}\, G$, $C_0 = G$, and $P_i = \emptyset$. Thus, $\mathcal{M}(P_0), C_0 \models \mathtt{EF}\, G$.
Case $i > 0$. The induction hypothesis is $\mathcal{M}(P_{i-1}), C_{i-1} \models \mathtt{EF}\, G$. By Lemma B.10, the induction hypothesis, and $P_{c_i} = \textsc{PreCompW}(C_{i-1})$, we get

$$\mathcal{M}(P_{c_i} \cup P_{i-1}), \textsc{States}(P_{c_i}) \cup C_{i-1} \models \mathtt{EF}\, G$$

which is equal to $\mathcal{M}(P_i), C_i \models \mathtt{EF}\, G$. □

Theorem B.5 (Soundness) $\textsc{Weak}$ *is sound.*

Proof. If $\textsc{Weak}(s_0, G)$ returns a solution π after iteration i then $\pi = P_i$ and $s_0 \in C_i$. Thus by Lemma B.11, $\mathcal{M}(\pi), s_0 \models \mathtt{EF}\, G$. □

Lemma B.12 *For* $\textsc{Weak}(s_0, G)$*, we have* $s \in Q_i \Leftrightarrow \textsc{WDist}(s, G) = i$.

Proof. By induction on i.
Case $i = 0$. By definition of Q_0 and $\textsc{WDist}$, $s \in Q_0 \Leftrightarrow s \in G \Leftrightarrow \textsc{WDist}(s, G) = 0$.
Case $i > 0$. The induction hypothesis is $s \in Q_j \Leftrightarrow \textsc{WDist}(s, G) = j$ for $j < i$.
"$\Rightarrow$": Assume that $s \in Q_i$. By definition of Q_i and $\textsc{PreCompW}$, we get

$$s \in \textsc{States}(\{\langle s', a' \rangle \; : \; s' \notin C_{i-1} \wedge \textsc{Next}(s', a') \cap C_{i-1} \neq \emptyset\})$$

. Thus by the induction hypothesis and Lemma B.1, $\textsc{WDist}(s, G) \leq i$. However, if $\textsc{WDist}(s, G) = k < i$ then $s \in Q_k$ according to the induction hypothesis which is in conflict with the assumption that $s \in Q_i$. Thus, $\textsc{WDist}(s, G) = i$.
"$\Leftarrow$": Assume $\textsc{WDist}(s, G) = i$. Then by definition of $\textsc{WDist}$, $\exists s', a \,.\, s' \in \textsc{Next}(s, a) \wedge \textsc{WDist}(s') = i - 1$. Thus, by the induction hypothesis and Lemma B.1, $s' \in C_{i-1}$. It must be the case that $s \notin C_{i-1}$ since $s \in C_{i-1}$ contradicts the induction hypothesis. But then by definition of $\textsc{PreCompW}$, $s \in \textsc{States}(\textsc{PreCompW}(C_{i-1}))$ which by definition of Q_i gives $s \in Q_i$. □

Theorem B.6 (Completeness) $\textsc{Weak}$ *is complete.*

Proof. If a valid solution exists then $\textsc{WDist}(s_0, G) \neq \infty$. Assume $\textsc{WDist}(s_0, G) = k$. Then by definition of $\textsc{WDist}$ there exists $k + 1$ states $q_0, \cdots, q_k$ such that $\textsc{WDist}(q_i) = i$ for $0 \leq i \leq k$, $q_k = s_0$, and $q_0 \in G$. Thus by Lemma B.12, $Q_i \neq \emptyset$ for $0 \leq i \leq k$, and $s_0 \in Q_k$. Since $Q_i = \textsc{States}(P_{c_i})$, we have $P_{c_i} \neq \emptyset$ for $1 \leq i \leq k$. Thus, $\textsc{Weak}(s_0, G)$ will not terminate with failure in the first k iterations. Also, it will not terminate with success in the first $k - 1$ iterations, since by Lemma B.2 $s_0 \notin Q_i$ for $0 \leq i < k$. However, since $s_0 \in Q_k$, it will terminate with success in iteration k. □

Lemma B.13 *For* $\text{WEAK}(s_0, G)$*, we have* $\forall s \in Q_i \,.\, \text{MIN}(s, G, P_i) = i$.

Proof. By induction on i.
Case $i = 0$. Let $s \in Q_0$.By definition of Q_0, we have $s \in G$. But then by definition of MIN, $\text{MIN}(s, G, \emptyset) = \text{MIN}(s, G, P_0) = 0$.
Case $i > 0$. The induction hypothesis is, $\forall s \in Q_j \,.\, \text{MIN}(s, G, P_j) = j$ for $j < i$. Let $s \in Q_i$. Then by definition of Q_i and $\text{PRECOMPW}(C_{i-1})$,

$$s \in \text{STATES}(\{\langle s', a' \rangle \,:\, s' \notin C_{i-1} \wedge \text{NEXT}(s', a') \cap C_{i-1} \neq \emptyset\}).$$

Thus, there exists an action a such that $\langle s, a \rangle \in P_{c_i}$ and $\text{NEXT}(s, a) \cap C_{i-1} \neq \emptyset$. Since $P_i = P_{i-1} \cup P_{c_i}$, we get from the induction hypothesis that $\text{MIN}(s, G, P_i) \leq i$. However, $\text{MIN}(s, G, P_i) < i$ implies $\text{WDIST}(s, G) < i$ which contradicts Lemma B.12 since we assume that $s \in Q_i$. Thus, $\text{MIN}(s, G, P_i) = i$. □

Theorem B.7 (Optimality) *If* π *is a solution returned by* $\text{WEAK}(s_0, G)$ *then*

$$\text{MIN}(s_0, G, \pi) = \text{WDIST}(s_0, G).$$

Proof. Combining Lemma B.12 and Lemma B.13 gives

$$\forall s \in Q_i \,.\, \text{MIN}(s, G, P_i) = \text{WDIST}(s, G)$$

which implies the result. □

B.6 Strong Cyclic

Lemma B.14 PRECOMPSC *is a valid precomponent function.*

Proof. By inspection of $\text{PRECOMPSC}(C)$ (l.4) it follows that if $\langle s, a \rangle \in \text{PRECOMPSC}(C)$ then $a \in \text{APP}(s)$ and $s \notin C$. To prove that $\text{PRECOMPSC}(C)$ terminates, we first observe that PRUNEOUTGOING terminates since it consists of a single preimage computation. PRUNEUNCONNECTED must also terminate since $NewSA$ clearly grows in each iteration and the number of SAs is finite. To prove that SCPLANAUX terminates, we have just shown that PRUNEOUTGOING and PRUNEUNCONNECTED terminates. By definition of PRUNEOUTGOING and PRUNEUNCONNECTED it is clear that $SA_{i+1} \subseteq SA_i$ in SCPLANAUX. Thus, as long as the loop in SCPLANAUX continues, we have $SA_{i+1} \subset SA_i$. Since the number of SAs is finite, SCPLANAUX eventually must terminate. PRECOMPSC must terminate since every call to SCPLANAUX terminates and wSA grows in each iteration. Thus, PRECOMPSC can only complete a finite number of iterations since the number of SAs is finite. □

Lemma B.15 *In each iteration of* PRUNEUNCONNECTED, *we have*

$$\mathcal{M}(NewSA_i), \text{STATES}(NewSA_i) \models \texttt{EF}\, C.$$

Proof. By induction on i.
Case $i = 0$. $NewSA_0 = \emptyset$ which trivially fulfills the requirement.
Case $i > 0$ The induction hypothesis is $\mathcal{M}(NewSA_{i-1}), \text{STATES}(NewSA_{i-1}) \models \texttt{EF}\, C$. We have

$$\begin{aligned} NewSA_i &\subseteq \text{PREIMGSA}(C \cup \text{STATES}(NewSA_{i-1})) \\ &\subseteq \{\langle s, a\rangle : \text{NEXT}(s, a) \cap (C \cup \text{STATES}(NewSA_{i-1})) \neq \emptyset\}. \end{aligned}$$

This means $\mathcal{M}(NewSA_i), \text{STATES}(NewSA_i) \models \texttt{EF}\, (C \cup \text{STATES}(NewSA_{i-1}))$. Combined with the induction hypothesis, we get $\mathcal{M}(NewSA_i \cup NewSA_{i-1}), \text{STATES}(NewSA_i) \cup \text{STATES}(NewSA_{i-1}) \models \texttt{EF}\, C$. Since clearly $NewSA_i \supseteq NewSA_{i-1}$, we have

$$\mathcal{M}(NewSA_i), \text{STATES}(NewSA_i) \models \texttt{EF}\, C.$$

$\square$

Lemma B.16 *If* SCPLANAUX$(startSA, C)$ *returns* π *then* $\mathcal{M}(\pi), \text{STATES}(\pi) \models \texttt{AGEF}\, C$

Proof. By inspection of SCPLANAUX, we have

$$\pi = \text{PRUNEUNCONNECTED}(\text{PRUNEOUTGOING}(\pi, c), c).$$

By definition of PRUNEOUTGOING, if $\langle s, a\rangle \in \pi$ then $\langle s, a\rangle \notin \text{PREIMGSA}(\overline{C \cup \text{STATES}(\pi)})$. Thus, if $s' \in \overline{C \cup \text{STATES}(\pi)}$ then $\langle s, s'\rangle \notin R$ of $\mathcal{M}(\pi)$. this proves that any execution path in $\text{EXEC}(q_0, \pi)$ where $q_0 \in \text{STATES}(\pi)$ can not reach a state outside of $C \cup \text{STATES}(\pi)$. However, we still need to prove that there is an execution path reaching C for each state in π. Since π is returned from PRUNEUNCONNECTED, we have $\pi = NewSA_i$ for some iteration of PRUNEUNCONNECTED. From Lemma B.15, we then get $\mathcal{M}(\pi), \text{STATES}(\pi) \models \texttt{AGEF}\, C$. $\square$

Lemma B.17 *For* STRONGCYCLIC(s_0, G), *we have* $\mathcal{M}(P_i), C_i \models \texttt{AGEF}\, G$.

Proof. By induction on i.
Case $i = 0$. We have $\mathcal{M}(\emptyset), G \models \texttt{AGEF}\, G$, $C_0 = G$, and $P_0 = \emptyset$. Thus, $\mathcal{M}(P_0), C_0 \models \texttt{AGEF}\, G$.
Case $i > 0$. The induction hypothesis is $\mathcal{M}(P_{i-1}), C_{i-1} \models \texttt{AGEF}\, G$. By Lemma B.16, $\mathcal{M}(P_{c_i}), \text{STATES}(P_{c_i}) \models \texttt{AGEF}\, C_{i-1}$. Combined with the induction hypothesis this gives $\mathcal{M}(P_{c_i} \cup P_{i-1}), \text{STATES}(P_{c_i}) \cup C_{i-1} \models \texttt{AGEF}\, G$. Which is equal to $\mathcal{M}(P_i), C_i \models \texttt{AGEF}\, G$.
$\square$

Theorem B.8 (Soundness) STRONGCYCLIC *is sound.*

Proof. If STRONGCYCLIC(s_0, G) returns a solution π after iteration i then $\pi = P_i$ and $s_0 \in C_i$. Thus by Lemma B.17, $\mathcal{M}(\pi), s_0 \models \texttt{AGEF}\, G$. $\square$

Lemma B.18 *If* $\mathcal{M}(\pi), s_0 \models \texttt{AGEF}\, G$ *and* $G \subseteq C$ *then*

$$M(\pi \setminus C \times Act), s_0 \models \texttt{AGEF}\, C.$$

Proof. Clearly $\mathcal{M}(\pi), s_0 \models \texttt{AGEF}\, C$ since $G \subseteq C$. In addition, we can remove SAs within C from π since they are not necessary to fulfill $\mathcal{M}(\pi), s_0 \models \texttt{AGEF}\, C$. Thus,

$$M(\pi \setminus C \times Act), s_0 \models \texttt{AGEF}\, C.$$

$\square$

Lemma B.19 *If* $\mathcal{M}(\pi), s_0 \models \texttt{AGEF}\, C$ *then there exists a* $\pi' \subseteq$ FIXEDPOINT(C) *where* $\mathcal{M}(\pi'), s_0 \models \texttt{AGEF}\, C$.

Proof. Let P denote the prefixes of any execution path in EXEC(s_0, π) that starts in s_0 and ends in a state in C. Let π' denote the SAs associated with the paths in P. We have $\mathcal{M}(\pi'), s_0 \models \texttt{AGEF}\, C$ since otherwise there would exist an execution path in EXEC(s_0, π) reaching a state from which C is unreachable. By definition of PREIMGSA, we have that FIXEDPOINT(C) contains any SA associated with any finite path that starts from a state in $\overline{C}$ and ends at a state in C. Thus, due to the definition of π', we have $\pi' \subseteq$ FIXEDPOINT(C). $\square$

Theorem B.9 (Completeness) STRONGCYCLIC *is complete.*

Proof. By contradiction. Assume that $\mathcal{M}(\pi), s_0 \models \texttt{AGEF}\, G$, but STRONGCYCLIC$(s_0, G)$ terminates in iteration i with "no solution exists". The ith call to PRECOMPSC is finding a strong cyclic precomponent of C_{i-1}. Since STRONGCYCLIC terminates in iteration i, we must have that PRECOMPSC$(C_{i-1}) = \emptyset$. Since wSA in PRECOMPSC(C_{i-1}) is updated in the same way as F in the FIXEDPOINT algorithm, there must exist an iteration k of PRECOMPSC(C_{i-1}) where $wSA_k =$ FIXEDPOINT(C_{i-1}). Let $\pi' = \pi \setminus C_{i-1} \times Act$. Since $G \subseteq C_{i-1}$, Lemma B.18 gives that $\mathcal{M}(\pi'), s_0 \models \texttt{AGEF}\, C_{i-1}$. But then by Lemma B.19, there exists a set of SAs $\pi'' \subseteq$ FIXEDPOINT$(C_{i-1}) = wSA_k$ such that $\mathcal{M}(\pi''), s_0 \models \texttt{AGEF}\, C_{i-1}$. Consider the pruning of wSA_k in SCPLANAUX. According to the proof of Lemma B.19, π'' can be chosen such that it has no SAs leading out from STATES$(\pi'') \cup C_{i-1}$, and any SA in π'' is associated with an execution path connected to C_{i-1}. Thus, no SAs will be pruned from π'' by PRUNEOUTGOING and PRUNEUNCONNECTED. Consequently, SCPLANAUX returns a non-empty result which in turn causes PRECOMPSC(C_{i-1}) to return a non-empty result, which is impossible. $\square$

B.7 GNDP

For all guided non-deterministic planning algorithms, we will assume that there are n partitions of the disjunctive branching partitioning used by the algorithms. Further, recall that for a map M, M denotes the union of the entries in M. To simplify the presentation, we assume that for a queue Q, the symbol Q both denotes the queue and the union of the entries in it.

Lemma B.20 *if C_i exists then $C_i \supset C_{i-1}$.*

Proof. If C_i is computed then by line 6-7, $C_i = C_{i-1} \cup \textsc{States}(P_{c_i})$ where $\textsc{States}(P_{c_i}) \neq \emptyset$ (otherwise GNDP terminates in line 4). By definition of valid guided precomponents, we have $\textsc{States}(P_{c_i}) \cap C_{i-1} = \emptyset$. Thus, $C_i \supset C_{i-1}$. □

Theorem B.10 (Termination) GNDP *terminates.*

Proof. By definition of valid guided precomponents, the function GPRECOMP called by GNDP must terminate. By Lemma B.20, we have that $C_i \supset C_{i-1}$ after completion of iteration i. However, since the state space is finite the number of iterations must also be finite. □

B.8 Guided Strong

Lemma B.21 *For* GPRECOMPS$(\mathbf{C})$*, we have*

$$\textsc{PreCompS}(C) = \bigcup_{j=1}^{|\mathbf{C}|} \bigcup_{i=1}^{n} \textsc{PreCompS}_i(C, \mathbf{C}[h_j])$$

Proof. By definition of PRECOMPS$_i$,

$$\bigcup_{i=1}^{n} \textsc{PreCompS}_i(C, \mathbf{C}[h_j]) = \bigcup_{i=1}^{n} (\textsc{PreImgSA}_i(\mathbf{C}[h_j]) \setminus \textsc{PreImgSA}(\overline{C})) \setminus C \times Act.$$

Thus, by definition of PREIMGSA and the fact that a disjunctive partitioning contains all the transitions of the transition relation

$$\begin{aligned}
\bigcup_{i=1}^{n} \textsc{PreCompS}_i(C, \mathbf{C}[h_j]) &= \bigcup_{i=1}^{n} \{\langle s, a\rangle : \langle s, a\rangle \in \textsc{PreImgSA}_i(\mathbf{C}[h_j]) \land \\
&\quad \langle s, a\rangle \notin \textsc{PreImgSA}(\overline{C}) \land s \notin C\} \\
&= \{\langle s, a\rangle : \langle s, a\rangle \in \textsc{PreImgSA}(\mathbf{C}[h_j]) \land \\
&\quad \langle s, a\rangle \notin \textsc{PreImgSA}(\overline{C}) \land s \notin C\}
\end{aligned}$$

From this we get

$$\bigcup_{j=1}^{|\mathbf{C}|}\bigcup_{i=1}^{n}\text{PRECOMPS}_i(C,\mathbf{C}[h_j]) =$$

$$\{\langle s,a\rangle \,:\, \langle s,a\rangle \in \bigcup_{j=1}^{|\mathbf{C}|}\text{PREIMGSA}(\mathbf{C}[h_j]) \wedge \langle s,a\rangle \notin \text{PREIMGSA}(\overline{C}) \wedge s \notin C\} =$$

$$\{\langle s,a\rangle \,:\, \langle s,a\rangle \in \text{PREIMGSA}(C) \wedge \langle s,a\rangle \notin \text{PREIMGSA}(\overline{C}) \wedge s \notin C\} =$$

$$\text{PRECOMPS}(C).$$

□

Lemma B.22 GPRECOMPS *is a valid guided precomponent function.*

Proof. By Lemma B.21, Q contains a partitioning of a strong precomponent. For each partition, the correct h-value $h = h_j - \delta h_i$ is associated with the states by INSERT. Since a map with the top node of Q is returned by GPRECOMPS, the output from GPRECOMPS has the correct form. Finally, we have that GPRECOMPS terminates, since all subcomputations terminates and the loops are finite. □

Lemma B.23 *If* $\mathbf{P_c} = \text{GPRECOMPS}(\mathbf{C})$ *then* $\mathcal{M}(P_c), \text{STATES}(P_c) \models \texttt{AF}\, C$.

Proof. From Lemma B.21, we have that $P_c \subseteq \text{PRECOMPS}(C)$. But then it follows from the proof of Lemma B.5 that $\mathcal{M}(P_c), \text{STATES}(P_c) \models \texttt{AF}\, C$. □

Lemma B.24 *For* $\text{GUIDEDSTRONG}(s_0, G)$*, we have* $\mathcal{M}(P_i), C_i \models \texttt{AF}\, G$.

Proof. Similar to Lemma B.6 when using Lemma B.23 instead of Lemma B.5. □

Theorem B.11 (Soundness) GUIDEDSTRONG *is sound.*

Proof. If $\text{GUIDEDSTRONG}(s_0, G)$ returns a solution π after iteration i then $\pi = P_i$ and $s_0 \in C_i$. Thus by Lemma B.24, $\mathcal{M}(\pi), s_0 \models \texttt{AF}\, G$. □

Lemma B.25 *If* $C \supseteq \bigcup_{t=0}^{k-1} SD_t(W)$ *and* $SD_k(W) \setminus C \neq \emptyset$ *then* $\text{PRECOMPS}(C) \neq \emptyset$.

Proof. Let $s \in SD_k(W) \setminus C$. Since $s \in SD_k(W)$, there exists an action a where

- $\langle s,a\rangle \in \text{PREIMGSA}(C) \supseteq \text{PREIMGSA}(\bigcup_{t=0}^{k-1} SD_t(W))$,
- $\langle s,a\rangle \notin \text{PREIMGSA}(\overline{C}) \subseteq \text{PREIMGSA}(\overline{\bigcup_{t=0}^{k-1} SD_t(W)})$.

Since also $s \notin C$, we get by definition of PRECOMPS that $s \in$ PRECOMPS(C). Thus PRECOMPS$(C) \neq \emptyset$. □

Lemma B.26 *If $SD_k(G) \neq \emptyset$ and* GUIDEDSTRONG(s_0, G) *returns "no solution exists" then there exists an iteration i of* GUIDEDSTRONG *where $\bigcup_{t=0}^{k} SD_t(G) \subseteq C_i$.*

Proof. By induction on k.
Case $k = 0$. By definition of SDIST, $SD_0(G) = G$. Since $C_0 = G$, we have $i = 0$.
Case $k > 0$. The induction hypothesis is that if $SD_{k-1}(G) \neq \emptyset$ and GUIDEDSTRONG(s_0, G) returns "no solution exists" then there exists an iteration i' of GUIDEDSTRONG where $\bigcup_{t=0}^{k-1} SD_t(G) \subseteq C_{i'}$.

Assume $SD_k(G) \neq \emptyset$. Then by definition of SDIST, $SD_{k-1}(G) \neq \emptyset$. Thus, by the induction hypothesis, GUIDEDSTRONG will not terminate before an iteration i' where $\bigcup_{t=0}^{k-1} SD_t(G) \subseteq C_{i'}$.

Consider an iteration $j \geq i'$. Let $R_j = SD_k(G) \setminus C_j$ denote the states in $SD_k(G)$ not covered by the plan. By Lemma B.25 and Lemma B.21, the queue in GPRECOMPS can only be empty if $R_j = \emptyset$. Thus at some iteration $i \geq i'$ before GUIDEDSTRONG terminates with "no solution exists", it must be the case that $\bigcup_{t=0}^{k} SD_t(G) \subseteq C_i$. □

Theorem B.12 (Completeness) GUIDEDSTRONG *is complete.*

Proof. By contradiction. Assume a solution exists, but GUIDEDSTRONG(s_0, G) terminates with failure. Since a solution exists, we have SDIST$(s_0, G) \neq \infty$. Assume SDIST$(s_0, G) = k$. We then have $SD_k(G) \neq \emptyset$. Thus, by Lemma B.26 GUIDEDSTRONG will continue to an iteration i where $SD_k(G) \subseteq C_i$. Since $s_0 \in SD_k(G)$ this will cause GUIDEDSTRONG to terminate with success, which is impossible. □

B.9 Guided Weak

Lemma B.27 *For* GPRECOMPW$(\mathbf{C})$*, we have*

$$\text{PRECOMPW}(C) = \bigcup_{j=1}^{|\mathbf{C}|} \bigcup_{i=1}^{n} \text{PRECOMPW}_i(C, \mathbf{C}[h_j])$$

Proof. Similar to Lemma B.21. □

Lemma B.28 GPRECOMPW *is a valid guided precomponent function.*

Proof. By Lemma B.27, Q contains a partitioning of a weak precomponent. For each partition, the correct h-value $h = h_j - \delta h_i$ is associated with the states by INSERT. Since a map with the top node of Q is returned by GPRECOMPW, the output from GPRECOMPW has the correct form. Finally, we have that GPRECOMPW terminates, since all subcomputations terminates and the loops are finite. □

Lemma B.29 *If* $\mathbf{P_c} = \text{GPRECOMPW}(\mathbf{C})$ *then* $\mathcal{M}(P_c), \text{STATES}(P_c) \models \texttt{EF}\, C$.

Proof. From Lemma B.27, we have that $P_c \subseteq \text{PRECOMPS}(C)$. But then it follows from the proof of Lemma B.10 that $\mathcal{M}(P_c), \text{STATES}(P_c) \models \texttt{EF}\, C$. □

Lemma B.30 *For* GUIDEDWEAK(s_0, G)*, we have* $\mathcal{M}(P_i), C_i \models \texttt{EF}\, G$.

Proof. Similar to Lemma B.11 when using Lemma B.29 instead of Lemma B.10. □

Theorem B.13 (Soundness) GUIDEDWEAK *is sound.*

Proof. If GUIDEDWEAK(s_0, G) returns a solution π after iteration i then $\pi = P_i$ and $s_0 \in C_i$. Thus by Lemma B.30, $\mathcal{M}(\pi), s_0 \models \texttt{EF}\, G$. □

Lemma B.31 *If* $C \supseteq \bigcup_{t=0}^{k-1} WD_t(W)$ *and* $WD_k(W) \setminus C \neq \emptyset$ *then* $PreCompW(C) \neq \emptyset$.

Proof. Let $s \in WD_k(W) \setminus C$. Since $s \in WD_k(W)$, there exists an action a where

$$\langle s, a \rangle \in \text{PREIMGSA}(C) \supseteq \text{PREIMGSA}(\bigcup_{t=0}^{k-1} WD_t(W))$$

Since also $s \notin C$, we get by definition of PRECOMPW that $s \in \text{PRECOMPW}(C)$. □

Lemma B.32 *If* $WD_k(G) \neq \emptyset$ *and* GUIDEDWEAK(s_0, G) *returns "no solution exists" then there exists an iteration* i *of* GUIDEDWEAK *where* $\bigcup_{t=0}^{k} WD_t(G) \subseteq C_i$.

Proof. By induction on k.
Case $k = 0$. By definition of WDIST, $WD_0(G) = G$. Since $C_0 = G$, we have $i = 0$.
Case $k > 0$. The induction hypothesis is that if $WD_{k-1}(G) \neq \emptyset$ and GUIDEDWEAK(s_0, G) returns "no solution exists" then there exists an iteration i' of GUIDEDWEAK where $\bigcup_{t=0}^{k-1} WD_t(G) \subseteq C_{i'}$.

Assume $WD_k(G) \neq \emptyset$. Then by definition of WDIST, $WD_{k-1}(G) \neq \emptyset$. Thus, by the induction hypothesis, GUIDEDWEAK will not terminate before an iteration i' where $\bigcup_{t=0}^{k-1} WD_t(G) \subseteq C_{i'}$.

Consider an iteration $j \geq i'$. Let $R_j = WD_k(G) \setminus C_j$ denote the states in $WD_k(G)$ not covered by the plan. By Lemma B.31 and Lemma B.27, the queue in GPRECOMPW can only be empty if $R_j = \emptyset$. Thus at some iteration $i \geq i'$ before GUIDEDWEAK terminates with "no solution exists", it must be the case that $\bigcup_{t=0}^{k} WD_t(G) \subseteq C_i$. □

Theorem B.14 (Completeness) GUIDEDWEAK *is complete.*

Proof. By contradiction. Assume a solution exists, but GUIDEDWEAK(s_0, G) terminates with failure. Since a solution exists, we have WDIST$(s_0, G) \neq \infty$. Assume WDIST(s_0, G) $= k$. We then have $WD_k(G) \neq \emptyset$. Thus, by Lemma B.32 GUIDEDWEAK will continue to an iteration i where $WD_k(G) \subseteq C_i$. Since $s_0 \in WD_k(G)$ this will cause GUIDEDWEAK to terminate with success, which is impossible. □

B.10 Guided Strong Cyclic

Lemma B.33 *If* $|Q| = 0$ *in iteration* i *of the repeat loop of* GPRECOMPSC(**C**) *then* $wSA_i =$ FIXEDPOINT(C).

Proof.
"$\subseteq$". Assume that $\langle s, a\rangle \in wSA_i$. By following the parent nodes in the search tree implicitly represented by Q, we get that $\langle s, a\rangle$ lies on a path $sp'p'' \cdots$ connected to C where $s \xrightarrow{a} p'$. Thus, by definition of FIXEDPOINT(C), $\langle s, a\rangle \in$ FIXEDPOINT(C).
"$\supseteq$". Assume $\langle s, a\rangle \in$ FIXEDPOINT(C). Thus, $\langle s, a\rangle$ lies on a path connected to C. Assume without loss of generality that this path is $p_m \cdots p_0$ where $p_m = s$ and $p_0 \in C$. Before the first iteration of the repeat loop, Lemma B.27 gives, $q_1 \in$ STATES(Q). But then Q can not be empty before q_1 is expanded. If $q_2 \notin$ STATES(wSA) before q_1 is expanded then q_2 is inserted in Q when q_1 is expanded. Otherwise another node has inserted q_2 in Q before q_1 was expanded. In both cases, Q can not be empty before q_2 is inserted. Applying this argument inductively, we get that Q is not empty before p_{m-1} has been inserted. Thus, at some point before iteration i of the repeat loop, $\langle s, a\rangle \in wSA$. □

Lemma B.34 GPRECOMPSC *is a valid guided precomponent function.*

Proof. Lemma B.33 gives that GPRECOMPSC(C) returns a map with valid SAs of C. By inspection of GPRECOMPSC, we see that the INSERT function associates states in Q with their correct h-value. Thus, all states in STATES(wSA) are associated with their correct h-value by **wS** in line 13. By inspection of line 21 to 22, we therefore get that the map of SAs returned by GPRECOMPSC(C) associates the states of the SAs with their correct h-value.

To prove that $\text{GPRECOMPSC}(C)$ terminates, we first observe that line 1 to 6 terminates since the loops are finite. The repeat loop must also terminate, since in the proof of Lemma B.33 wSA reaches a max size at some point, due to the finite number of SAs, such that no new nodes are inserted on Q. □

Theorem B.15 (Soundness) GUIDEDSTRONGCYCLIC *is sound.*

Proof. This follows from Lemma B.16 and an adoption of Lemma B.17. □

Theorem B.16 (Completeness) GUIDEDSTRONGCYCLIC *is complete.*

Proof. By contradiction. Assume that $\mathcal{M}(\pi), s_0 \models \texttt{AGEF}\, G$, but $\text{GUIDEDSTRONGCYCLIC}(s_0, G)$ terminates in iteration i with "no solution exists". The ith call to GPRECOMPSC is finding a strong cyclic precomponent of C_{i-1}. Since GUIDEDSTRONGCYCLIC terminates in iteration i, we must have $\text{GPRECOMPSC}(C_{i-1}) = \emptyset$. This is only possible if $|Q| = 0$ in some iteration of the repeat loop. But then by Lemma B.33, we have that prior to this $wSA = \text{FIXEDPOINT}(C_{i-1})$. We have shown in the completeness proof of STRONGCYCLIC that this leads to $\text{SCPLANAUX}(wSA) \neq \emptyset$, which is impossible since GPRECOMPSC then would return with a non-empty map. □

B.11 Weak Adversarial

Lemma B.35 PRECOMPWA *is a valid precomponent function.*

Proof. This follows from $\text{PRECOMPWA}(C) \subseteq \text{PREIMGSSA}(C) \setminus C \times Act_s$ and the fact that both PREIMGSSA and FAIRSTATES terminates. □

Lemma B.36 *If π_s is returned by* $\text{PRECOMPWA}(C)$ *then*

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(\pi_s, \pi_e), \text{STATES}_s(\pi_s) \models \texttt{EF}\, C.$$

Proof. By definition of $\text{PRECOMPWA}(C)$, we have

$$\begin{aligned} \pi_s \;=\; & \{\langle s, a_s\rangle : \langle s, a_s\rangle \in \text{PREIMGSSA}(C) \wedge s \notin C \wedge \\ & \forall a_e \in \text{APP}_e(s) \,.\, \exists a_s \in \text{ACT}_s(\text{PREIMGSSA}(C) \setminus C \times Act_s, s), s' \in C \,.\, s \xrightarrow{a_s, a_e} s'\}. \end{aligned}$$

Let $s \in \text{STATES}_s(\pi_s)$ and let $\pi_e \in \Pi_e^+$. Then by definition of Π_e^+, we have $\emptyset \subset \text{ACT}_e(\pi_e, s) \subseteq \text{APP}_e(s)$. Assume $a_e \in \text{ACT}_e(\pi_e, s)$. Then by definition of π_s there exists a counter system action a_s such that $\langle s, a_s\rangle \in \pi$ and $s \xrightarrow{a'_s, a_e} c$ where $c \in C$. By definition of R of $\mathcal{M}(\pi_s, \pi_e)$, we get $\langle s, c\rangle \in R$. But then clearly $\mathcal{M}(\pi_s, \pi_e), s \models \texttt{EF}\, C$. □

Lemma B.37 *For* WEAKADVERSARIAL(s_0, G)*, we have*

$$\forall \pi_e \in \Pi_e^+ . \mathcal{M}(P_i, \pi_e), C_i \models \mathtt{EF}\, G.$$

Proof. By induction on i.
Case $i = 0$. For any $\pi_e \in \Pi_e^+$, we have $\mathcal{M}(\emptyset, \pi_e), s \models \mathtt{EF}\, G$ if $s \in G$. Thus, since $C_0 = G$ and $P_0 = \emptyset$, we get $\forall \pi_e \in \Pi_e^+ . \mathcal{M}(P_0, \pi_e), C_0 \models \mathtt{EF}\, G$.
Case $i > 0$. The induction hypothesis is $\forall \pi_e \in \Pi_e^+ . \mathcal{M}(P_{i-1}, \pi_e), C_{i-1} \models \mathtt{EF}\, G$. From Lemma B.36 and $P_{c_i} = \text{PRECOMPWA}(C_{i-1})$, we get $\forall \pi_e \in \Pi_e^+ . \mathcal{M}(P_{c_i}, \pi_e), \text{STATES}_s(P_{c_i}) \models \mathtt{EF}\, C_{i-1}$. Thus, by the induction hypothesis, it must hold that $\forall \pi_e \in \Pi_e^+ . \mathcal{M}(P_{c_i} \cup P_{i-1}, \pi_e), \text{STATES}_s(P_{c_i}) \cup C_{i-1} \models \mathtt{EF}\, G$, which is equal to $\forall \pi_e \in \Pi_e^+ . \mathcal{M}(P_i, \pi_e), C_i \models \mathtt{EF}\, G$. □

Theorem B.17 (Soundness) WEAKADVERSARIAL *is sound.*

Proof. If WEAKADVERSARIAL(s_0, G) returns a solution π_s after iteration i then $\pi_s = P_i$ and $s_0 \in C_i$. Thus, by Lemma B.37 $\forall \pi_e \in \Pi_e^+ . \mathcal{M}(\pi_s, \pi_e), s_0 \models \mathtt{EF}\, G$. □

Lemma B.38 *If there exists a system plan* π_s *where*

$$\forall \pi_e \in \Pi_e^+ . \mathcal{M}(\pi_s, \pi_e), s \models \mathtt{EF}\, C$$

then there exists a weak marked adversarial DAG $AD_w(s, C)$*.*

Proof. If $s \in C$ then $AD_w(s, C)$ is the single terminal state s. Otherwise assume $s \notin C$. Since C can be reached from s for any non-empty environment plan there must exist a set of finite prefixes P of execution paths in $\bigcup_{\pi_e \in \Pi_e^+} \text{EXEC}(s, \pi_s, \pi_e)$ that reaches a state in C. In addition, P can be chosen such that

1. from each state v on a path in P visited prior to a state in C, a counter system action for each applicable environment action $\text{APP}_e(v)$ exists, and
2. all system actions associated with paths in P are counter actions.

The first property is fulfilled since no non-empty environment plan exists making C unreachable. The second property holds since, in order for C to be reachable, it is sufficient only to apply a single counter action for each environment action that may transition to a state closer to C.

Let the level of a state q on one of these paths be defined by the maximum distance from q to C for any of the paths visiting q. The paths then form a weak marking $AD_w(s, C)$ of an adversarial DAG $AD(C)$ of C. □

Theorem B.18 (Completeness) WEAKADVERSARIAL *is complete.*

Proof. By contradiction. Assume that a system plan π_s exists such that $\forall \pi_e \in \Pi_e^+ . \mathcal{M}(\pi_s, \pi_e), s_0 \models \texttt{EF}\, G$ and WEAKADVERSARIAL(s_0, G) terminates with "no solution exists". By Lemma B.38, a weak marked adversarial DAG $AD_w(s_0, G)$ exists. By the definition of $AD_w(s_0, G)$ we have that all states at level i are fair with respect to their applicable system actions and the states at level $j < i$. Thus, since PRECOMPWA prunes unfair states from the preimage of C, all states in $AD_w(s_0, G)$ at level 1 are in C_1, all states at level 2 are in C_2 etc.. Hence, if C has reached a maximum size (which must happen since WEAK terminates with failure) then C includes all states in $AD_w(s_0, G)$. However, then WEAKADVERSARIAL returns with success since $s_0 \in$ WEAKADVERSARIAL(s_0, G), which is impossible. □

B.12 Strong Cyclic Adversarial

Lemma B.39 *Let S_i be defined recursively by*

$$\begin{aligned} S_0 &= \emptyset\,, \\ S_i &= SA \cap \textsc{FairStates}(SA, C \cup \textsc{States}_s(S_{i-1})) \times Act_s. \end{aligned}$$

then $S_{i+1} \supseteq S_i$.

Proof. By induction on i.
Case $i = 0$. Trivial since $S_0 = \emptyset$.
Case $i > 0$. The induction hypothesis is $S_i \supseteq S_{i-1}$. We have

$$S_{i+1} = SA \cap \textsc{FairStates}(SA, C \cup \textsc{States}_s(S_i)) \times Act_s.$$

Thus, by definition of FAIRSTATES

$$\begin{aligned} S_{i+1} &= SA \cap \{s : \forall a_e \in \textsc{App}_e(s)\,.\,\exists a_s \in \textsc{Act}_s(SA, s), \\ &\quad s' \in C \cup \textsc{States}_s(S_i)\,.\, s \xrightarrow{a_s, a_e} s'\} \times Act_s. \end{aligned}$$

The induction hypothesis is $S_i \supseteq S_{i-1}$ which means

$$\begin{aligned} S_{i+1} &\supseteq SA \cap \{s : \forall a_e \in \textsc{App}_e(s)\,.\,\exists a_s \in \textsc{Act}_s(SA, s), \\ &\quad s' \in C \cup \textsc{States}_s(S_{i-1})\,.\, s \xrightarrow{a_s, a_e} s'\} \times Act_s \\ &= S_i. \end{aligned}$$

□

Lemma B.40 PRECOMPSCA *is a valid precomponent function.*

Proof. By inspection of $PreCompSCA(C)$, we have

$$PreCompSCA(C) \subseteq \text{FIXEDPOINT}(C).$$

Thus, if $\langle s, a\rangle \in$ PRECOMPSCA(C) then $a \in \text{APP}(s)$ and $s \notin C$. To prove that PRECOMP-$SCA(C)$ terminates, we observe that the only difference between PRECOMPSC(C) and PRECOMPSCA(C) is that the subfunction PRUNEUNCONNECTED has been substituted with PRUNEUNFAIR. We can therefore reuse the proof for termination of PRECOMPSC(C) except that we need to show that PRUNEUNFAIR terminates. Assume that PRUNEUNFAIR diverges. By Lemma B.39 $NewSA_{i+1} \supseteq NewSA_i$. However, since PRUNEUNFAIR diverges, we must have $NewSA_{i+1} \supset NewSA_i$ for $i > 0$, which is impossible since the number of SSAs is finite. □

Lemma B.41 *In each iteration i of* PRUNEUNFAIR(SA, C)*, we have*

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(NewSA_i, \pi_e), \text{STATES}_s(NewSA_i) \models \text{EF}\, C.$$

Proof. By induction on i.
Case $i = 0$. $NewSA_0 = \emptyset$ which trivially fulfills the requirement.
Case $i > 0$. The induction hypothesis is

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(NewSA_{i-1}, \pi_e), \text{STATES}_s(NewSA_{i-1}) \models \text{EF}\, C.$$

We have

$$NewSA_i \;=\; SA \cap \text{FAIRSTATES}(SA, C \cup \text{STATES}_s(NewSA_{i-1})) \times Act_s$$

Thus by definition of FAIRSTATES

$$\begin{aligned} NewSA_i \;=\;& SA \cap \{s : \forall a_e \in \text{APP}_e(s) \,.\, \exists a_s \in \text{ACT}_s(SA, s), \\ & s' \in C \cup \text{STATES}_s(NewSA_{i-1}) \,.\, s \xrightarrow{a_s, a_e} s'\} \times Act_s. \end{aligned}$$

Let $s \in \text{STATES}_s(NewSA_i)$ then obviously

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(NewSA_i, \pi_e), s \models \text{EF}\, C \cup \text{STATES}(NewSA_{i-1}).$$

Thus,

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(NewSA_i, \pi_e), \text{STATES}_s(NewSA_i) \models \text{EF}\, C \cup \text{STATES}(NewSA_{i-1}).$$

Combined with the induction hypothesis, we get $\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(NewSA_i \cup NewSA_{i-1}, \pi_e)$, $\text{STATES}_s(NewSA_i) \cup \text{STATES}_s(NewSA_{i-1}) \models \text{EF}\, C$. Thus, since by Lemma B.39 $NewSA_i \supseteq NewSA_{i-1}$

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(NewSA_i, \pi_e), \text{STATES}_s(NewSA_i) \models \text{EF}\, C.$$

□

Lemma B.42 *If* $\text{SCAP}\textsc{lan}\text{A}\textsc{ux}(startSA, C)$ *returns* π_s *then*

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(\pi_s, \pi_e), \textsc{States}(\pi_s) \models \texttt{AGEF}\, C.$$

Proof. By inspection of SCAPLANAUX, we have

$$\pi_s = \textsc{PruneUnfair}(\textsc{PruneOutgoing}(\pi_s, C), C).$$

By definition of PRUNEOUTGOING, if $\langle s, a_s \rangle \in \pi_s$ then

$$\langle s, a_s \rangle \notin \textsc{PreImg}\text{SSA}(\overline{C \cup \textsc{States}_s(\pi_s)}).$$

Thus, for all $\pi_e \in \Pi_e^+$ if $s' \in \overline{C \cup \textsc{States}_s(\pi_s)}$ then $\langle s, s' \rangle \notin R$ of $\mathcal{M}(\pi_s, \pi_e)$. This means that no execution path in $\textsc{Exec}(q_0, \pi_s, \pi_e)$ where $q_0 \in \textsc{States}_s(\pi_s)$ can reach a state outside of $C \cup \textsc{States}_s(\pi_s)$. We still need to show that for all $\pi_e \in \Pi_e^+$ there is an execution path reaching C for each state in π_s. However, this follows from Lemma B.41, since there exists an i such that $\pi_s = NewSA_i$ of PRUNEUNFAIR. From the above, it follows

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(\pi_s, \pi_e), \textsc{States}_s(\pi_s) \models \texttt{AGEF}\, C.$$

□

Lemma B.43 *For* $\textsc{StrongCyclicAdversarial}(s_0, G)$*, we have*

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(P_i, \pi_e), C_i \models \texttt{AGEF}\, G.$$

Proof. By induction on i.
Case $i = 0$. We trivially have

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(P_0, \pi_e), g \models \texttt{AGEF}\, G$$

for $g \in G$. Thus,

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(P_0, \pi_e), C_0 \models \texttt{AGEF}\, G.$$

Case $i > 0$. The induction hypothesis is

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(P_{i-1}, \pi_e), C_{i-1} \models \texttt{AGEF}\, G.$$

From Lemma B.42 and $P_{c_i} = \textsc{PreComp}\text{SCA}(C_{i-1})$, we get

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(P_{c_i}, \pi_e), \textsc{States}_s(P_{c_i}) \models \texttt{AGEF}\, C_{i-1}.$$

Combined with the induction hypothesis, we get

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(P_{c_i} \cup P_{i-1}, \pi_e), \textsc{States}_s(P_{c_i}) \cup C_{i-1} \models \texttt{AGEF}\, G.$$

Which is equal to

$$\forall \pi_e \in \Pi_e^+ \,.\, \mathcal{M}(P_i, \pi_e), C_i \models \texttt{AGEF}\, G.$$

□

Theorem B.19 (Soundness) STRONGCYCLICADVERSARIAL *is sound.*

Proof. If STRONGCYCLICADVERSARIAL(s_0, G) returns a solution π_s after iteration i then $\pi_s = P_i$ and $s_0 \in C_i$. Thus, by Lemma B.43 $\forall \pi_e \in \Pi_e^+ . \mathcal{M}(\pi_s, \pi_e), s_0 \models \texttt{AGEF}\, G$. $\square$

Lemma B.44 *If there exists a system plan π_s where*

$$\forall \pi_e \in \Pi_e^+ . \mathcal{M}(\pi_s, \pi_e), s \models \texttt{AGEF}\, C$$

then there exists a strong cyclic marked adversarial DAG $AD_{sc}(s, C)$.

Proof. If $s \in C$ then $AD_{sc}(s, C)$ is the single terminal state s. Otherwise, assume $s \notin C$. Let V denote the set of states that can be visited by any execution path in $\bigcup_{\pi_e \in \Pi_e^+} \text{EXEC}(s, \pi_s, \pi_e)$ prior to a state in C. For each state in $v \in V$, we have $\forall \pi_e \in \Pi_e^+ . \mathcal{M}(\pi_s, \pi_e), v \models \texttt{AGEF}\, C$. Thus, it must be possible to define a counter system action for each applicable environment action $\text{APP}_e(v)$ such that a set of finite paths reaching C exists where the states of the paths are in V and the associated system actions are counter actions. Let the level of a state q on one of these paths be defined by the maximum distance from q to C for any of the paths visiting q. The paths can then be represented by a DAG which is a subset of an adversarial DAG $AD(C)$ of C. Since the states of this DAG is V and $s \in V$, a strong cyclic marking of the DAG exists that is equal to a valid strong cyclic marking $AD_{sc}(s, C)$ of $AD(C)$. $\square$

Lemma B.45 *If a strong cyclic marked adversarial DAG $AD_{sc}(s, C)$ exists and $AD_{sc}^{SA}(s, C) \subseteq SA$ then $AD_{sc}^{SA}(s, C) \subseteq$* PRUNEUNFAIR$(SA, C)$.

Proof. By inspection of PRUNEUNFAIR(SA, C), it follows that all SSAs of a state s in SA is included in $NewSA_{i+1}$ if s is fair with respect to the SSAs associated with s and the states in $NewSA_i$. Since all the states at level 0 in $AD_{sc}^{SA}(s, C)$ are in C, we get that all SSAs of states at level 1 are included in $NewSA_1$. Similarly, in iteration 2, we get that all SSAs of states at level 2 are included in $NewSA_2$ and PRUNEUNFAIR(SA, C) does not terminate in iteration 1 if some states at level 2 are not in $\text{STATES}_s(NewSA_1)$. Thus, by induction, an iteration k is reached where $AD_{sc}^{SA}(s, C) \subseteq NewSA_k$. $\square$

Theorem B.20 (Completeness) STRONGCYCLICADVERSARIAL *is complete.*

Proof. By contradiction. Assume that $\forall \pi_e \in \Pi_e^+ . \mathcal{M}(\pi_s, \pi_e), s_0 \models \texttt{AGEF}\, G$, but STRONG-CYCLICADVERSARIAL(s_0, G) terminates in iteration i with "no solution exists". The ith call to PRECOMPSCA is finding a strong cyclic adversarial precomponent of C_{i-1}. Since STRONGCYCLICADVERSARIAL terminates in iteration i, we must have PRECOMPSCA$(C_{i-1}) = \emptyset$. Since $\forall \pi_e \in \Pi_e^+ . \mathcal{M}(\pi_s, \pi_e), s_0 \models \texttt{AGEF}\, G$, we also have

$$\forall \pi_e \in \Pi_e^+ . \mathcal{M}(\pi_s, \pi_e), s_0 \models \texttt{AGEF}\, C_{i-1}$$

since by Lemma B.3 $C_{i-1} \supseteq G$. Thus, by Lemma B.44 a strong cyclic marked adversarial DAG $AD_{sc}(s_0, C_{i-1})$ exists. We have from the completeness proof of STRONGCYCLIC that in some iteration of $\text{PRECOMPSCA}(C_{i-1})$, $wSA = \text{FIXEDPOINT}(C_{i-1})$. From the definition of $AD_{sc}(s_0, C_{i-1})$, it is clear that $AD^{SA}_{sc}(s_0, C_{i-1}) \subseteq \text{FIXEDPOINT}(C_{i-1})$. Consider the pruning of wSA in SCAPLANAUX. No SSAs in $AD^{SA}_{sc}(s_0, C_{i-1})$ will be pruned by PRUNEOUTGOING since $AD^{SA}_{sc}(s_0, C_{i-1})$ has no SSAs leading out from $\text{STATES}(AD^{SA}_{sc}(s_0, C_{i-1})) \cup C_{i-1}$. In addition, by Lemma B.45 no SSAs in $AD^{SA}_{sc}(s_0, C_{i-1})$ will be pruned by $\text{PRUNEUNFAIR}(SA, C_{i-1})$ since $AD^{SA}_{sc}(s_0, C_{i-1}) \subseteq SA$. Thus SCAPLANAUX and PRECOMPSCA return a non-empty result, which is impossible. □