# Certifying Planning Systems: Witnesses for Unsolvability

**Inauguraldissertation**
zur
Erlangung der Würde eines Doktors der Philosophie
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel

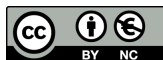
von

Salomé Eriksson

aus Reutigen BE

2019

Originaldokument gespeichert auf dem Dokumentenserver der Universität Basel
edoc.unibas.ch




Dieses Werk ist unter einer Creative Commons Lizenz vom Typ Namensnennung-Nicht kommerziell 4.0 International zugänglich. Um eine Kopie dieser Lizenz einzusehen, konsultieren Sie http://creativecommons.org/licenses/by-nc/4.0

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

Prof. Dr. Malte Helmert,
Universität Basel, Dissertationsleiter, Fakultätsverantwortlicher

Prof. Dr. Fahiem Bacchus
University of Toronto, Korreferent

Basel, den 23.04.2019

Prof. Dr. Martin Spiess,
Universität Basel, Dekan

*To Gunnar and my parents*

# Abstract

Classical planning tackles the problem of finding a sequence of actions that leads from an initial state to a goal. Over the last decades, planning systems have become significantly better at answering the question whether such a sequence exists by applying a variety of techniques which have become more and more complex. As a result, it has become nearly impossible to formally analyze whether a planning system is actually correct in its answers, and we need to rely on experimental evidence.

One way to increase trust is the concept of *certifying algorithms*, which provide a *witness* which justifies their answer and can be verified independently. When a planning system finds a solution to a problem, the solution itself is a witness, and we can verify it by simply applying it. But what if the planning system claims the task is unsolvable? So far there was no principled way of verifying this claim.

This thesis contributes two approaches to create witnesses for unsolvable planning tasks. *Inductive certificates* are based on the idea of invariants. They argue that the initial state is part of a set of states that we cannot leave and that contains no goal state. In our second approach, we define a *proof system* that proves in an incremental fashion that certain states cannot be part of a solution until it has proven that either the initial state or all goal states are such states.

Both approaches are complete in the sense that a witness exists for every unsolvable planning task, and can be verified efficiently (in respect to the size of the witness) by an independent verifier if certain criteria are met. To show their applicability to state-of-the-art planning techniques, we provide an extensive overview how these approaches can cover several search algorithms, heuristics and other techniques. Finally, we show with an experimental study that generating and verifying these explanations is not only theoretically possible but also practically feasible, thus making a first step towards *fully certifying* planning systems.

# Zusammenfassung

Klassische Handlungsplanung befasst sich mit dem Problem, eine Folge von Aktionen zu finden, welche von einem Startzustand zu einem Ziel führt. Planungssysteme sind in den letzten Jahrzehnten immer besser darin geworden die Frage nach der Existenz einer solchen Sequenz zu beantworten, indem sie eine Vielzahl von immer komplexeren Techniken anwenden. Dadurch wurde es aber fast unmöglich, die Korrektheit solcher Systeme formal zu analysieren.

*Zertifizierende Algorithmen* sind eine Möglichkeit, Vertrauen in einen Algorithmus zu erhöhen. Sie rechtfertigen ihre Antwort durch ein zusätzlich generiertes *Zertifikat*, welches unabhängig vom Algorithmus verifiziert werden kann. Falls ein Planungssystem eine Lösung für ein Problem findet, ist die Lösung selbst ein Zertifikat, welches durch Anwendung verifiziert werden kann. Aber was, wenn das Planungssystem sagt, es gibt keine Lösung? Bis jetzt gab es kein formales Verfahren, diese Aussage zu verifizieren.

Diese Dissertation stellt zwei Varianten zur Generierung von Zertifikaten unlösbarer Probleme in der Handlungsplanung vor. *Induktive Zertifikate* basieren auf der Idee von Invarianten. Sie zeigen Unlösbarkeit indem sie eine Menge von Zuständen finden, welche man nicht verlassen kann und welche den Anfangs-, aber keinen Zielzustand beinhaltet. Als zweite Variante stellen wir ein *Beweissystem* vor, welches schrittweise zeigt, dass gewisse Teile des Problems nicht Teil einer Lösung sein können, bis der Beweis erbracht ist, dass dies für den Anfangs- oder alle Zielzustände gilt.

Bei beiden Varianten kann für jedes unlösbare Planungsproblem ein Zertifikat erstellt werden, welches unter gewissen Umständen effizient (in Bezug zur Grösse des Zertifikats) und unabhängig verifiziert werden kann. Indem wir die Erstellung solcher Zertifikate für eine Vielzahl von Suchalgorithmen, Heuristiken und anderen Planungstechniken vorstellen, zeigen wir auf, dass sie für heute gängige Planungssysteme geeignet sind. Schliesslich demonstrieren wir anhand einer experimentellen Evaluation, dass es nicht nur theoretisch sondern auch praktisch möglich ist, diese Zertifikate zu generieren und zu verifizieren, und machen dadurch einen ersten Schritt zu einem *vollständig verifizierenden* Planungssystem.

# Acknowledgments

Writing this thesis has been an incredible endeavor and would not have been possible without the support from many wonderful people in my life.

First I would like to thank my advisor Malte Helmert. His enthusiasm about his research was immediately apparent in his lectures and was what drew me towards academia. I am grateful not only for giving me the possibility to graduate in his research group but also for his support during my time here. I continue to be amazed by his incredible knowledge, his enthusiasm to share this knowledge and his dedication to — despite his tremendous workload — always take time to extensively answer any questions I had.

I also want to thank Fahiem Bacchus for agreeing to join my thesis committee, even if this involves flying around half the world in order to attend my defense; and for the fruitful discussions we had when he visited our group two years ago.

Working in the AI research group in Basel has been both incredibly productive and enjoyable, thanks to my amazing colleagues: Patrick Ferber, Guillem Frances, Cedric Geissmann, Manuel Heusner, Thomas Keller, Florian Pommerening, Gabi Röger, Jendrik Seipp, Silvan Sievers and Martin Wehrle. Thanks for all the helpful discussions, as well as for our off-topic chats and leisure moments, for instance our board game nights.

My thanks also go to Jörgen Rosèn, Johan Karlborg, Gabi Röger, Jan Simon, Katharina Spreyermann and Gunnar Eriksson for proofreading parts of my thesis, and Sarah Simon for helping me with the layout and making the plots look amazing.

I am of course also very grateful for all the people in my social environment who continuously supported me. I would like to thank all my friends for providing sometimes much needed distraction, and my family for always being by my side. Thanks to my brother Jan, for planting the idea in my head to study computer science in the first place. To my sister Sarah, for listening and sharing experiences during those panic moments before an all to close deadline. To my parents Katharina Spreyermann and Markus Simon, for giving me the possibility to follow my dreams and supporting me no matter what. To my grandparents Marianna and Hans Simon for planning ahead for the future of their grandchildren, and my grandmother Elsa Spreyermann for being an inspiration, graduating during a time where this was uncommon for women.

Finally I want to thank my husband Gunnar Eriksson. For listening when I got all excited upon a new insight in my research and could not stop rambling about it. For being patient with me when I was stressed. For always being there for me. And for so much more!

# Contents


**1 Introduction** **1**
1.1 Contributions . . . 2
1.2 Outline . . . 3
1.3 Relation to Published Work . . . 3
1.4 Experimental Setup . . . 4

**2 Related Work** **5**
2.1 Automated Theorem Proving . . . 5
2.2 Boolean Satisfiability Problem . . . 6
2.2.1 DPLL and Resolution Proofs . . . 7
2.2.2 CDCL and (D)RUP proofs . . . 8
2.2.3 Preserving Satisfiability and (D)RAT Proofs . . . 9
2.3 Model Checking . . . 10
2.4 Classical Planning . . . 11

**3 Background** **13**
3.1 Classical Planning . . . 13
3.1.1 $SAS^+$ Planning Tasks . . . 15
3.2 Representation of State Sets . . . 17
3.2.1 Operations . . . 19
3.2.2 Specific Formalisms . . . 20

**I Witnesses** **28**

**4 Inductive Certificates** **29**
4.1 Inductive Sets . . . 29
4.1.1 Backwards Inductive Sets . . . 31
4.2 Simple Inductive Certificates . . . 32
4.3 Disjunctive Certificates . . . 35
4.4 Conjunctive Certificates . . . 37

**5 Unsolvability Proof System** **40**
5.1 Natural Deduction . . . 40
5.2 Proof System for Unsolvability of Planning Tasks . . . 42
5.2.1 Inference Rules . . . 42

5.2.2 Basic Statements . . . 46
5.2.3 Overview . . . 47
5.3 Efficient Verification . . . 49
5.3.1 Single Representation . . . 50
5.3.2 Mixed Representations . . . 53
5.4 Comparison to Inductive Certificates . . . 56

**II Applications** **62**

**6 Search Algorithms** **63**
6.1 Blind Search . . . 63
6.1.1 Explicit Blind Search . . . 64
6.1.2 Symbolic Blind Search . . . 65
6.2 Heuristic Search . . . 65
6.2.1 Inductive Certificates . . . 66
6.2.2 Proof System . . . 68

**7 Heuristics** **71**
7.1 Delete Relaxation . . . 72
7.2 Critical Paths . . . 73
7.3 Merge and Shrink . . . 75
7.4 Landmarks . . . 79

**8 Other Approaches** **80**
8.1 Trapper . . . 80
8.2 Iterative Dead Pairs Calculation . . . 82
8.3 Clause-Learning State Space Search . . . 84

**III Experimental Evaluation** **85**

**9 Inductive Certificates** **86**
9.1 Implementation . . . 87
9.2 Generation . . . 88
9.3 Verification . . . 90

**10 Proof System** **93**
10.1 Implementation . . . 94
10.2 Generation . . . 95
10.3 Verification . . . 97

**11 Comparison** **99**
11.1 Witness Size . . . . . . . . . . 99
11.2 Generation . . . . . . . . . . 99
11.3 Verification . . . . . . . . . . 101
11.4 Summary . . . . . . . . . . 102

**IV Future Work and Conclusion** **104**

**12 Future Work** **105**
12.1 Witnesses . . . . . . . . . . 105
12.2 Applications . . . . . . . . . . 106

**13 Conclusion** **109**

**Appendix** **111**

**Appendix A Proof Details** **112**
A.1 Complexity of BDD Operations . . . . . . . . . . 112
A.2 Translation of Inductive Certificates to Proofs . . . . . . . . . . 114

**Appendix B Detailed Coverage Results** **120**

**Bibliography** **122**

# 1 Introduction

In planning we try to find a sequence of actions that enables us to achieve a predefined goal from our current situation. For example, the logistic problem of distributing packages from one location to another can be framed as a planning problem: our current situation, called the *initial state* describes where all the packages and delivery trucks currently are, our goal describes where the packages should end up, and our actions consist of loading packages into a truck, driving a truck and unloading packages at a location. In this thesis we focus on *classical* planning where we have full knowledge about our current situation and actions are *deterministic*, *discrete* and *fully observable*, meaning if we apply an action to our current situation we know exactly how it changes.

Traditionally, research in classical planning has mostly focused on finding solutions as fast as possible. If a solution has been found, it is usually emitted and we can verify it by applying it step by step and see if this results in achieving our goal. But what if the problem has no solution? Currently, planning systems simply emit variations of the phrase "no solution found" if they are unable to find one, but how can we know that no solution exists, i.e. that the planning system is correct?

One way of providing correctness guarantees is to prove the correctness of the entire planning system with formal theorem provers. However, due to the complexity of state-of-the-art planning systems this approach is often infeasible. The current practice is to provide a high-level description of the algorithm used in the planning system and prove correctness of this high-level description. While this increases trust, it does not guarantee that a concrete implementation is correct. Empirical testing of the implementation on problems where the answers is known further increases trust but again does not give any guarantees.

A more promising approach is turning planning systems into *certifying algorithms* (McConnell et al., 2011). The core idea of certifying algorithms is to emit a *witness* alongside the answer, which serves as a proof of the answer's correctness and can be verified independently by a so-called *verifier*. While this does not guarantee that the certifying algorithm is correct on *all* inputs, it guarantees that the answer for a concrete input is correct if the witness is validated by the independent checker (and if the checker itself is correct). Planning systems nowadays are already partially certifying since they output a plan if they find one, which in itself serves as a witness. But if the task is unsolvable, they do not produce any form of witness.

The work presented in this thesis makes a first step in the direction of fully certifying planning systems by investigating how such witnesses could be defined, produced by planning systems, and verified.

## 1.1 Contributions

We propose two types of witnesses for unsolvable planning tasks:

- **Inductive certificates**. This type of witness describes a property that is preserved through action application and that the initial state satisfies but no goal state does, thus showing that no path from the initial state to any goal can exist.
- **Proofs**. We define a *proof system* that focuses on proving sets of states *dead*, i.e. proving that they cannot be part of any solution. A proof showing that the initial state or all goal states are dead then serves as a witness.

To analyze their suitability, we use the following properties as a guideline:

- **soundness and completeness**: If a witness for a planning task exists, the task must be unsolvable; and for every unsolvable planning task we can form a witness.
- **efficient generation**: Generating a witness for a concrete problem should incur no higher than polynomial overhead for the planning system.
- **efficient verification**: The time complexity for verifying the correctness of a witness should be at most polynomial in its size.
- **generality**: It should be possible for a wide range of planning systems to generate a witness.

These properties often result in trade-offs as to which property is valued more. For example, if a witness is very general, it might incur more overhead to the planning system than if we used a witness that is specifically designed for exactly this planning system. Another property in this trade-off that plays an important role in efficient verification is the size of the witness, which can vary significantly between different witness types. While we do not optimize for witness size and allow for sizes exponential in the problem description, we bound it in respect to the runtime of the planner by requiring efficient generation.

We show for both types of witnesses that they are sound and complete and under which conditions efficient verification is possible. Furthermore, we demonstrate their generality by describing how a variety of planning systems can be altered to emit such a witness, and that in these cases generation is efficient as well.

Complementing our theoretical contributions, we implemented verifiers for both types of witnesses, as well as augmented several configurations of the Fast Downward planning system (Helmert, 2006) to emit them. These practical contributions are evaluated in an empirical study, which showcases that both types of witnesses can feasibly be emitted and verified.

## 1.2 Outline

Before discussing our main contribution, Chapter 2 performs a literature survey detailing how other areas in computer science increase trust in their algorithms. Chapter 3 then formally introduces the classical planning problem, as well as several formalisms used in our witnesses.

Chapters 4 and 5 introduce inductive certificates and the proof system respectively, showing important theoretical results like completeness and conditions for efficient verification.

The following three chapters show how the witnesses can be used in current planning systems; with Chapter 6 focusing on different search algorithms, Chapter 7 on heuristics and Chapter 8 on other prominent planning techniques.

Afterwards, Chapters 9 and 10 perform an empirical study on the practical usage of inductive certificates and proofs individually, before comparing the two in Chapter 11.

Finally, Chapter 12 gives an outlook on ideas how the proposed witnesses could be enhanced and used to cover more planning techniques, before Chapter 13 concludes the thesis by summarizing the results of our contribution.

## 1.3 Relation to Published Work

A majority of the work presented in this thesis has been published in conference proceedings of major conferences on automated planning and artificial intelligence:

- The paper *Unsolvability Certificates for Classical Planning* (Eriksson, Röger, and Helmert, 2017) introduces inductive certificates and its variations. It then demonstrates the application of inductive certificates to blind search, heuristic search and the Trapper algorithm, and performs a preliminary experimental evaluation. Its contributions form a majority of Chapter 4, and are discussed in Chapters 6, 7 and 8.1. The experimental evaluation in Chapter 9 and 11 is based on the same implementation as was used in the paper.

  This paper won the **ICAPS 2017 best student paper award**.

- The paper *Inductive Certificates of Unsolvability for Domain-Independent Planning* (Eriksson, Röger, and Helmert, 2018a) was invited for submission to the Best Papers From Sister Conferences Track at IJCAI and is based on Eriksson, Röger, and Helmert (2017), providing a more accessible introduction to inductive certificates.

- The paper *A Proof System for Unsolvable Planning Tasks* (Eriksson, Röger, and Helmert, 2018b) introduces the proof system for unsolvable planning tasks. It examines the relationship of the proof system to inductive certificates and discusses its application to heuristic search with multiple heuristics, clause learning state-space

search and iterative dead pairs calculation. Finally, it also offers an experimental evaluation and comparison to inductive certificates. Contributions from this paper form a large part of Chapter 5, and are discussed in Chapter 6, 8.2 and 8.3. The implementation used in Chapters 10 and 11 is based on the one from the paper.

## 1.4 Experimental Setup

For our experimental study, we implemented a verifier for each type of witness presented. Furthermore, we augmented the Fast Downward planning system (Helmert, 2006) as well as the planning system used in Steinmetz and Hoffmann (2017) to emit witnesses for a selection of configurations. For the representation of Binary Decision Diagrams, we utilized the CUDD library (Somenzi, 2015). As a framework for running the experiments we used the Downward Lab toolkit (Seipp et al., 2017). The code for the two verifiers as well as for the augmented planning systems is publicly available (Eriksson, 2019a).

The planning problems used in the experiments originate from (a) the benchmarks used in the Unsolvability IPC 2016[1] and (b) Hoffmann, Kissmann, and Torralba (2014).[2] Since we are only interested in unsolvable problems, we removed all solvable problems from the benchmark set. Some domains and problems overlap: The problems in the two directories bottleneck and unsat-pegsol-strips from (b) are identical to the ones in bottleneck and pegsol from (a) and thus omitted. The resulting benchmark set is published online (Eriksson, 2019c). We assume that problems in folders unsat-nomystery, unsat-rovers and unsat-tpp from (b) also overlap with those in folders over-nomystery, over-rovers and over-tpp from (a). We kept all problems from those folders, but report them jointly under domain name nomystery, rovers and tpp. Furthermore we removed several tasks from diagnosis[3] from the experimental evaluation since they contained conditional effects, which we do not consider.

All experiments are run on a cluster consisting of Core Intel Xeon Silver 4114 processors with a clock speed of 2.2 GHz. For generating witnesses, the planning systems were given 30 minutes time and 3584 MiB memory. For verification, the same memory limit was used but the verifiers were given 4 hours of time. Both generation and verification use a single CPU core. All generated experiment data is available online (Eriksson, 2019b).

[1] `https://unsolve-ipc.eng.unimelb.edu.au` (accessed 18.01.2019).

[2] `http://fai.cs.uni-saarland.de/downloads/unsat-benchmarks.tar.bz2`, (accessed 18.01.2019).

[3] The diagnosis problems we omit are #07,08,10,12,15,17 and 19.

# 2 Related Work

While correctness is important in every area of computer science, the emphasis put on proving it varies widely. Almost every algorithm is at least proven correct in a hand-crafted proof for its high-level description, as it is invaluable not only for trust but also for understanding the algorithm itself. Concrete implementations of an algorithm on the other hand are usually only tested by means of unit tests and empirical evaluation on benchmarks. For many algorithms this level of correctness guarantee is already sufficient, especially if the benchmarks used in the empirical evaluation are diverse. For other algorithms, stronger guarantees are necessary, such as witnesses (proving correctness for a specific input) or a formal proof (proving correctness for any input).

In this chapter we investigate how different areas of computer science and specifically Artificial Intelligence handle correctness guarantees. We first cover *Automated Theorem Proving*, which is the strongest proof of correctness for an algorithm, and is employed in several research areas across all of computer science. We then study the *Boolean Satisfiability Problem*, a fundamental AI problem which influences almost all areas of AI research. Next, we examine *Model Checking*, an area related to planning whose focus it is to verify the correctness of hardware and software systems. Finally, we investigate existing work in proving correctness of *Classical Planning* algorithms, and highlight how the contributions of this thesis provide stronger correctness guarantees for planning systems.

## 2.1 Automated Theorem Proving

Automated Theorem Proving aims to automatically generate proofs for theorems within a given theory. A prover is built on an underlying logic in which the theory must be specified, and a deduction system which is used to reason about the theory. For example, if we define a theory based on propositional logic stating “it is day or it is night” and “it is not night”, a theorem prover with resolution as its deductive system can prove “it is day”.

The choice of the underlying logic and deduction system trades expressiveness versus efficiency or even ability of finding a proof. We showcase this here for three commonly used logics, propositional, first order (FOL) and higher order (HOL). For propositional logic, deduction systems exist that are sound (if a theorem is proven, it is valid) and complete (if a theorem is valid, it can be proven), and following Cook (1971) we know that deciding whether a proof exists is **co-NP**-complete (although concrete proofs might have exponential size). FOL is more expressive but even though deduction systems maintaining

soundness and completeness exist they are only semi-decidable (Gödel, 1929), i.e. while finding a proof for a valid theorem is guaranteed to terminate, the theorem prover might run forever on invalid theorems. Finally, HOL is the most expressive logic of the three, but according to Gödel's incompleteness theorem (Gödel, 1931) there can be no computable deduction system for it that is also sound and complete. However, this does not disqualify HOL as an underlying logic, since a sound but incomplete theorem prover can still find many proofs, just not all. HOL is in fact successfully used in several state-of-the-art theorem provers such as HOL4[1], Isabelle[2] or PVS[3].

A theorem prover attempts to find a proof by searching for a sequence of applicable inference rules that result in the desired theorem. This search can span an enormous space of logical inferences, possibly even an infinite space. To speed up the search, an *interactive theorem prover* enables the user to give additional input in the form of hints on which area the prover should focus. For example, one might specify which inference rules might be particularly useful, or which sub-theorems might be needed. An interactive theorem prover can also be used to verify the correctness of an existing proof by simply specifying all proof steps.

Automated theorem provers are used in both mathematics and computer science. They have for example been successfully used to prove the Robbins conjecture (McCune, 1997), several decades after the conjecture had been postulated but was never proven by humans. In computer science, a widespread commercial use is integrated circuit design (Russinoff, 2000), which became especially relevant after a bug in the floating point unit of Pentium Processors had been discovered in 1994. In terms of software verification, some examples include the verification of the OS kernel seL4 (Klein et al., 2009) or the verified C compiler CompCert[4].

## 2.2 Boolean Satisfiability Problem

The *Boolean Satisfiability Problem* (SAT) addresses the question if a given propositional formula can be satisfied by at least one assignment. It was the first problem shown to be **NP**-complete (Cook, 1971). As such, many hard problems like Hamilton paths, Vertex Covers and Graph Coloring can be reduced to SAT. More exactly, the proof shows that any **NP**-complete problem can be reduced to a special case of SAT where the formula is given in Conjunctive Normal Form (CNF). For this reason, research in this area usually only considers CNF formulas and we will restrict our discussion to this case as well.

SAT is an integral part in many areas of AI such as constraint programming and planning, and is also widely used in industrial settings such as railways, avionics and automotive (Hammarberg and Nadjm-Tehrani, 2005; Pěnička, 2007). In many of these areas it is

[1] `https://hol-theorem-prover.org` (accessed 01.02.19).
[2] `https://isabelle.in.tum.de` (accessed 01.02.19).
[3] `http://pvs.csl.sri.com` (accessed 01.02.19).
[4] `http://compcert.inria.fr/research.html` (accessed 31.01.2019).

vital that the used SAT solver is correct (i.e. does not give any wrong answers), necessitating some form of verification of the SAT solver.

While verified SAT solvers exist (e.g. Blanchette et al., 2018), they typically are developed from a theoretical perspective in order to formally verify general concepts in SAT. For practical applications in performance critical environments an optimized SAT solver is better suited. However, as an optimized implementation moves further away from theoretically proven pseudo code, bugs tend to occur and verification becomes more important. In this setting, a certifying SAT solver which generates a witness for its output can combine the best of both worlds with only marginal losses: having a proof (albeit only for a concrete output), and maintaining an optimized performance (albeit inducing some overhead for creating the witness).

Generating a witness for a positive answer in SAT is easy: A SAT solver reports a formula as satisfiable if it finds an assignment that satisfies the formula. This assignment directly serves as a witness for satisfiability, and can be verified efficiently by evaluating the formula under this assignment, which takes time linear in the formula size. Defining and generating a suitable witness for a negative answer (i.e. the formula is unsatisfiable) is harder. With the advancements of SAT solvers in the last decades, several proof formats have been developed, often augmenting or complementing previous ones to cover new arising techniques used in the solvers.

Once a witness has been generated, it needs to be checked by a separate verifier program. The most commonly used verifiers are normal algorithms, usually developed alongside the proof format itself. But since running a verifier is usually not as time critical and it is equally important that the verifier itself is not faulty, *verified* verifiers have also been implemented (e.g. Lammich, 2017). We will however focus our discussion here on the evolution of proof formats rather than concrete verifier implementations.

### 2.2.1 DPLL and Resolution Proofs

The most influential algorithm in SAT is the *Davis-Putnam-Logemann-Loveland algorithm* (DPLL). Its origins lie in a sub procedure of the Davis-Putnam algorithm (Davis and Putnam, 1960). While the Davis-Putnam algorithm was used for testing validity of first-order logic formulas, the purpose of the sub-procedure was to determine unsatisfiability of a propositional formula. Davis, Logemann, and Loveland (1962) replaced this sub-procedure with what eventually came to be known as the the DPLL algorithm, which is still today a core part of state-of-the-art SAT solvers.

The first technique of extracting a proof from a DPLL based SAT solver is based on the *resolution rule*. It states that if a CNF $\varphi$ formula contains two clauses $c_1 = (l_1 \vee \cdots \vee l_n \vee l)$ and $c_2 = (l'_1 \vee \cdots \vee l'_m \vee \neg l)$ where one variable occurs positively in one clause and negatively in the other, the clause $c = (l_1 \vee \cdots \vee l_n \vee l'_1 \vee \cdots \vee l'_m)$ is implied by $c_1 \wedge c_2$ and thus also by $\varphi$. When applying a resolution rule we say we *resolve* $c_1$ *and* $c_2$ *on* $l$.

A resolution proof (Gelder, 2002) shows that a CNF formula is unsatisfiable by repeatedly applying the resolution rule to original and already inferred clauses until the *empty*

*clause* can be derived by applying the rule to two clauses $l$ and $\neg l$. This proves unsatisfiability of the formula since a disjunction over zero elements corresponds to falsity, and if a formula implies falsity, the formula must be unsatisfiable.

For example, consider formula $\chi = (x \lor y \lor \neg z) \land (\neg x \lor \neg z) \land (x \lor \neg y) \land z$. We denote the four clauses by $c_1$ to $c_4$. Resolving $c_1$ with $c_2$ on $x$ yields $c_5 = (y \lor \neg z)$, and resolving $c_2$ with $c_3$ on $x$ results in $c_6 = (\neg y \lor \neg z)$. We now can resolve $c_5$ with $c_6$ on $y$, leading to $c_6 = (\neg z)$. Finally, resolving $c_6$ and $c_4$ on $z$ results in an empty clause, thus showing that $\chi$ is unsatisfiable.

Resolution proofs are complete in the sense that for any unsatisfiable CNF formula at least one resolution proof exists (Robinson, 1965). Additionally, they are easy to check in relation to their size. Two major drawbacks however are that proofs can grow very large and that generating a proof often requires major modifications to the solvers.

### 2.2.2 CDCL and (D)RUP proofs

The aforementioned drawbacks of resolution based proofs became even more pronounced with the rise of *conflict driven clause learning* (CDCL), the most influential augmentation of the DPLL algorithm to date. To combat the ever growing size of proofs and complicated changes to SAT solvers, Goldberg and Novikov (2003) introduced a new paradigm of verifying unsolvability with the help of *unit propagation*, a technique already used in the DPLL algorithm. Given a formula $\varphi$, unit propagation iteratively picks a clause consisting of only one literal $l$ and assigns the corresponding variable. It then simplifies the formula by removing all clauses containing $l$, and removes all occurrences of $\neg l$ in all clauses. The resulting formula together with the variable assignments is equivalent to $\varphi$.

Similar to resolution proofs, *reverse unit propagation* (RUP) proofs (Gelder, 2008) iteratively add new clauses implied by $\varphi$ until the empty clause is added. However, the new clause $c$ does not need to be the result of a resolution application. Instead the implication must be proven with unit propagation: If the unit propagation of $\varphi \land \neg c$ contains the empty clause, $\varphi$ must imply $c$[5]. As with resolution proofs, clauses that already have been proven to be implied by $\varphi$ are used to strengthen $\varphi$, which allows for later clauses to be verified with unit propagation.

While the unit propagation of $\varphi \land \neg c$ for a $c$ implied by $\varphi$ does not necessarily result in an empty clause, it is the case for all clauses obtained by resolution: given resolvent $c = A \lor B$ from clauses $c_1 = l \lor A$ and $c_2 = \neg l \lor B$, the unit propagation of $\neg c \land c_1 \land c_2$ will reduce $c_1$ to $l$ and $c_2$ to $\neg l$, and then either reduce $c_1$ or $c_2$ to the empty clause. Thus, any resolution proof can be emulated by a RUP proof, meaning RUP proofs are complete. Additionally, we can conclude that RUP proofs are at least as compact as resolution proofs.

[5]Note that unit propagation does not offer a complete implication check, i.e. a clause *not* proven to be implied $\varphi$ by unit propagation might still be implied by $\varphi$ nonetheless. A polynomial complete check can only exist if $\mathbf{P} = \mathbf{NP}$, since checking if the empty clause is implied by $\varphi$ is equivalent to answering the question whether $\varphi$ is unsatisfiable.

RUP proofs excel where resolution proofs struggle: both DPLL and CDCL allow for an easy extraction of the implied clauses relevant for the proof without major changes to the code. Furthermore, proof sizes are generally significantly smaller than corresponding resolution proofs. However, verifying a clausal proof is in contrast more expensive in relation to the proof size. To combat the cost of verification, *delete reverse unit propagation* (DRUP) proofs (Heule, Hunt, and Wetzler, 2013a) additionally allow to specify the deletion of input or learned clauses at any point, meaning unit propagation checks performed afterwards do not consider this clause anymore. This preserves correctness because if $\varphi \models c$ holds then $\varphi \wedge c' \models c$ must hold as well.

### 2.2.3 Preserving Satisfiability and (D)RAT Proofs

As a newer trend in SAT, techniques that alter the input formula such that only satisfiability (but not equivalence) is preserved became more popular. Both resolution and clausal proofs cannot directly support these techniques.

A proof that adds clauses that guarantee preservation of satisfiability is a valid way of proving unsatisfiability, but as with adding implied clauses, showing that a clause addition preserves satisfiability is in general **NP**-complete. However, Järvisalo, Heule, and Biere (2012) introduced a property called *resolution asymmetric tautology* (RAT) which can identify certain clauses whose addition preserves satisfiability and can be tested in polynomial time. Given a clause $c$ and formula $\varphi$, $c$ is RAT for $\varphi$ if either (a) reverse unit propagation on $\varphi \wedge \neg c$ results in the empty clause or (b) some literal $l$ in $c$ exists such that for all clauses $c'$ obtained by resolving $c$ with a clause from $\varphi$ on $l$, we have that RUP on $\varphi \wedge \neg c'$ results in the empty clause. Clauses added by RUP proofs all have the RAT property since they satisfy (a), and many new techniques not covered by RUP proofs can produce RAT clauses.

The RAT proof format (Heule, Hunt, and Wetzler, 2013b) is almost identical to RUP proofs in that each line of the proof specifies a new clause which is added to the input formula. If the clause has the RAT property through (b), the corresponding literal must be the first literal of the clause. Since clauses in RUP proofs all have the RAT property though (a), a RUP proof is thus directly also a RAT proof. As with RUP and DRUP, DRAT proofs allow for deletion of clauses in order to make verification more efficient.

Since RAT is a generalization of RUP, verification is in general more expensive in relation to the proof size for RAT than for RUP, but RAT proofs can instead be much smaller than RUP proofs. Furthermore, if a RAT verifier always first checks (a) and is given a RUP proof, performance will be similar to a RUP verifier.

**SAT competitions** Requiring proofs for unsatisfiability was first tested in an experimental track of the SAT competition in 2005, for which the resolution proof format (Gelder, 2002) was specifically designed. The track returned two years later but due to the huge certificate size of resolution proofs the newly designed RUP format was used. The track grew in the following years, often allowing different proof formats. In 2014 however, the

organizers decided to only support DRAT proofs, since all participants in previous years choose a variation of RUP and DRAT is backwards compatible with it. Finally, from 2016 onward proving unsatisfiability was no more confined to a special track, instead it became mandatory in all tracks to provide certificates for unsatisfiable formulas.

## 2.3 Model Checking

Model checking aims to verify that a given hardware or software system is correct in the sense that it is impossible for them to reach a property that is considered erroneous. As an example, consider a CPU with multiple cores sharing the same memory: A state where two CPUs would gain exclusive write access to the shared memory at the same time would be considered an *error state*. A system is defined by a description of its current state and a description on how this state can change over time. The system can also be seen as a graph where the nodes are all possible states in which the system can be and the edges show which changes to a state are possible and what they result in. Model checking is thus closely related to planning, as both problems span such a graph. The two areas complement each other in the sense that planning tries to find a path from the initial state to any goal, while model checking tries to confirm that no path from the initial state to any error state exists.

Model Checking in itself can be understood as a form of correctness guarantee for the system being verified. Similar to certifying algorithms and their verifiers however, applying model checking to a system can falsely guarantee correctness of the system if the model checking algorithm used is faulty. To alleviate this problem, there has been work on both implementing a fully verified and a certifying model checker.

One example of a *verified* model checker for systems specified in temporal logic is presented in Esparza et al. (2013). While their verified algorithm does not reach the performance of non-verified ones, they are still able to incorporate code optimizations through a so-called refinement framework. The main idea of this framework is to first prove that a non-optimized abstract algorithm is correct, and then perform optimization through refinement steps, where it is proven that each refinement step preserves correctness. This allows to run the model checker in the optimized version, but proving the correctness incrementally starting from a simpler abstract algorithm.

As an example of a *certifying* algorithm, Conchon, Mebsout, and Zaïdi (2015) have shown how a model checker for parameterized systems can be augmented to emit witnesses when no error state can be reached, without noticeably lowering performance on the model checker. The witness is a logical formula encoding states of the system such that (1) the initial state is a model of the formula, (2) applying a transition step to the formula implies the same formula and (3) the formula implies the safety property. The intuition behind these properties is that the formula describes an invariant, i.e. if the system is in a state satisfying the formula, it cannot reach a state that does not satisfy it. As we will see in Chapter 4, the idea is very similar to our definition of inductive certificates.

## 2.4 Classical Planning

In classical planning we try to find a sequence of actions (plan) that reaches a predefined goal from a given starting point. State-of-the-art planning systems are semi-certifying in the sense that if they find a plan they output it, but give no proof if they say a task is unsolvable. This is because historically, emphasis has almost exclusively been on finding plans for solvable problems, and verifying a plan given by the solver is faster and easier than verifying the entire algorithm. Planning has thus also mostly been applied to solvable problems; a fact also reflected by the International Planning Competitions (IPC): almost all problems contained in competitions up to 2014 are solvable, with unsolvable problems occurring rather by accident than by design.

For solvable planning instances, verifying whether the answer given by the planner is correct is fairly straightforward. Planners can usually output the plan with little to no overhead, and verifying the plan can be done in time polynomial in the size of the plan and the problem description[6] (i.e. a fraction of the time a planner normally spends on finding the plan) by applying it to the initial state and test if the result is a goal state. Plan validators such as VAL (Howey and Long, 2003) and INVAL (Haslum, 2017) are thus routinely used in both research papers and the IPC. Abdulaziz and Lammich (2018) recently also introduced a verified version of a plan validator, showing in the process that both VAL and INVAL still contained bugs which only occurred in very rare instances and thus remained undetected so far.

With the heavy focus on fast plan detection, state-of-the-art planners did not fare as well when confronted with unsolvable planning tasks. Highlighting this issue, Bäckström, Jonsson, and Ståhlberg (2013) introduced an algorithm specifically tailored to detect *unsolvability* as fast as possible, based on projecting the task on a subset of variables. While the presented algorithm is incomplete (i.e. it does not detect all unsolvable planning tasks as unsolvable), experimental evaluation showed that it outperformed several state-of-the-art planners when confronted with unsolvable tasks. Ståhlberg's PhD thesis (2017) further investigated how to find good projections.

Inspired by this new line of research, Hoffmann, Kissmann, and Torralba (2014) utilized the M&S framework (a generalization of variable projections) to build a heuristic tailored to detecting dead-end states (states from which no goal can be reached). Later, Steinmetz and Hoffmann (2017) tailored the $h^C$ heuristic to recognize and learn from previously encountered dead-ends. The IPC reacted by holding the first *unsolvability* IPC in 2016. As a result, a variety of planning techniques have been adapted to become more proficient at detecting unsolvable planning tasks, e.g. potentials heuristics (Seipp et al., 2016), decoupled search (Gnad et al., 2016) or property directed reachability (Balyo and Suda, 2016).

Certain techniques can be seen as a form of witness for unsolvability. For example, potential heuristics for unsolvability detection try to build a function that evaluates successor

[6] Note that this is only true for planning problems given in the STRIPS formalism, which we consider in this thesis.

states with a value lower or equal than the value of their predecessor, while ensuring that the value of the initial state is lower than the value of any goal state. The existence of such a function shows unsolvability, since we can never reach a state with a higher function value than the state we start from. The $h^C$ heuristic in Steinmetz and Hoffmann (2017) can provide a formula that describes dead-end states and is transitive in the sense that any descendant of a state satisfying the formula also satisfy it. When the heuristic learns to prove the initial state to be a dead-end, this formula can serve as a form of witness.

However, the above witnesses can only be generated by their specific method and do not offer a clear procedure for verification. In contrast, this thesis focuses on providing a formal framework applicable to a wide variety of planning techniques, where witnesses are defined by a number of properties that are proven to imply unsolvability and can be verified efficiently.

# 3 Background

The first part of this chapter formally defines classical planning tasks. While two different definitions are commonly used in research, we restrict our discussion to one and present a well known compilation from the other one to the one we use. The second part revolves around propositional formulas, which are used to encode sets of states in our witnesses. Aside from describing how state sets can be encoded, we discuss what types of operations different formalisms used in propositional logic can efficiently perform, which will be needed to analyze the efficiency of generating and verifying witnesses.

## 3.1 Classical Planning

A classical planning task informally consists of the initial state of the world, a selection of actions we can use to alter the world and a goal which we want to achieve. As an example, consider a Sokoban puzzle as shown in Figure 3.1. The goal of this task is to move the box from `C3` to `B4`, but we can only push the box, not pull it.

We consider the STRIPS formalism (Fikes and Nilsson, 1971), where the state of the world is described with the help of propositional variables. In our example Sokoban problem, we could for example have a variable `box-at-C3` denoting whether or not the box is currently at location `C3`. A *state* is then defined by the variables that are currently true:

**Definition 3.1** (state)**.** *Let $V$ be a set of propositional variables. A* state *is a subset $s \subseteq V$.*

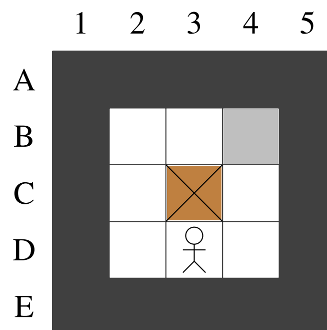



Figure 3.1: An example of the Sokoban puzzle. Dark gray cells are walls, the brown cell represents the box and the gray cell its goal position.

In our example Sokoban task, $V$ could consist of variables `box-at-X` and `man-at-X` for all grid positions X. The state depicted in Figure 3.1 would then be defined by the set $\{$`box-at-C3`, `man-at-D3`$\}$.

The goal description is also defined as a set of variables that must be true if we achieved our goal. For example the set $\{$`box-at-B4`$\}$ describes the goal of our Sokoban task.

**Definition 3.2** (goal states). *Let $V$ be a set of propositional variables and $G \subseteq V$ the goal description. A state $s \subseteq V$ is a* goal state *if $s \supseteq G$.*

An action in STRIPS alters the state by changing the value of certain variables. In the Sokoban task, we could define action `push-C3-B3` which would push the box to `B3` by setting `box-at-C3` to false and `box-at-B3` to true. Additionally, an action contains a list of preconditions which describes to which states an action can be applied to. For example we should only be able to apply `push-C3-B3` if the box is indeed currently at `C3` and the man stands in `D3`. Formally an action is defined as follows:

**Definition 3.3** (action). *An action is a tuple $a = \langle pre(a), add(a), del(a) \rangle$ where $pre(a) \subseteq V$ is the* precondition*, $add(a) \subseteq V$ the* add-effects *and $del(a) \subseteq V$ the* delete effects.

An action can be applied to a state if it satisfies all preconditions, and changes the state by adding the add effects, removing the delete effects and leaving everything else unaltered.

**Definition 3.4** (action application). *An action $a$ is* applicable *in state $s$ iff $pre(a) \subseteq s$. After applying $a$ to $s$, the resulting* successor state *$s[a]$ is defined as follows:*

$$s[a] = \begin{cases} (s \setminus del(a)) \cup add(a) & \text{if } pre(a) \subseteq s \\ \text{undef.} & \text{otherwise} \end{cases}$$

*A sequence of actions $\pi = \langle a_1 \dots a_n \rangle$ is called applicable in $s$ iff $s[a_1] \dots [a_i]$ is defined for all $1 \leq i \leq n$. We use the shorthand $s[\pi]$ to denote $s[a_1] \dots [a_n]$.*

We are now ready to give the full definition of a planning task:

**Definition 3.5** (STRIPS planning task). *A* STRIPS planning task *is defined as a tuple $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$, where*

- *$V^\Pi$ is a finite set of propositional variables*
- *$A^\Pi$ is a finite set of actions*
- *$I^\Pi \subseteq V^\Pi$ is the initial state*
- *$G^\Pi \subseteq V^\Pi$ is the goal description.*

*We write $\|\Pi\|$ for the size of a description of $\Pi$, $S^\Pi$ for the set of all states of $\Pi$ (i.e. the power set of $V^\Pi$) and $S^\Pi_G$ for the set of all goal states.*

Usually, planning tasks also contain a cost function denoting the cost of each action, where solutions with lower cost are better. However, since the cost function does not influence whether or not the problem is solvable we omit it in our definition.

A planning task implicitly induces a graph, where the nodes correspond to states and action applications form the edges between nodes. This graph is called the *search space*.

**Definition 3.6** (plan)**.** *Given a planning task* $\Pi$ *and state* $s$*, a sequence of actions* $\pi = \langle a_1, \ldots a_n \rangle$ *is called an* $s$-plan *if* $\pi$ *is applicable in* $s$ *and* $s[\pi] \supseteq G^\Pi$*. A* $I^\Pi$*-plan is also just called a plan.*

Since we will usually work with sets of states rather than single states, we also define action application to state sets:

**Definition 3.7** (progression)**.** *Given a planning task* $\Pi$*, state set* $S \subseteq S^\Pi$ *and action* $a \in A^\Pi$*,* $S[a] = \{s[a] \mid s \in S, a \text{ applicable in } s\}$ *is the* progression *of* $S$ *with* $a$*.*

*For a set of actions* $A$ *and state set* $S$*, the progression of* $S$ *with* $A$ *is defined as* $S[A] = \bigcup_{a \in A} S[a]$*. The progression of* $S$ *with all actions* $A^\Pi$ *of the planning task is also just called the* progression *of* $S$*.*

The progression of $S$ with $A$ is *monotonic* in $S$ and $A$, i.e., $S[A] \subseteq S'[A']$ for all $S \subseteq S'$ and $A \subseteq A'$.

Actions can also be applied backwards to a state set $S$, meaning we calculate all predecessor states from which you can reach a state in $S$ with the chosen actions:

**Definition 3.8** (regression)**.** *Given a planning task* $\Pi$*, state set* $S \subseteq S^\Pi$ *and action* $a \in A^\Pi$*,* $[a]S = \{s' \mid a \text{ applicable in } s', s'[a] \in S\}$ *is the* regression *of* $S$ *with* $a$*.*

*For a set of actions* $A$ *and state set* $S$*, the regression of* $S$ *with* $A$ *is defined as* $[A]S = \bigcup_{a \in A}[a]S$*. The regression of* $S$ *with all actions* $A^\Pi$ *of the planning task is also just called the regression of* $S$*.*

As for progression, regression is monotonic in $S$ and $A$.

### 3.1.1 SAS$^+$ Planning Tasks

The SAS$^+$ planning formalism (Bäckström and Nebel, 1995) is the other commonly used formalism for planning. Contrary to STRIPS, the variables in SAS$^+$ are not binary but each variable $v$ has an associated finite domain *dom*$(v)$. For the above Sokoban task for example, we could define a variable `box` with a domain denoting all grid positions. A *fact* is a tuple consisting of a variable and a value of its domain, such as $\langle \texttt{box}, \texttt{B2} \rangle$ which denotes that the box is at `B2`. States are defined as follows:

**Definition 3.9** (SAS$^+$ states)**.** *Given a set of variables* $\mathcal{V}$*, a partial function* $p : \mathcal{V} \to \bigcup_{v \in \mathcal{V}} dom(v)$ *assigning variables to values in their respective domain is called a* partial state*. The variables assigned in a partial state* $p$ *are denoted as vars*$(p)$*.*

*A* state $s$ *is a function that assigns all variables in* $\mathcal{V}$*.*

A SAS$^+$ task is now defined as follows:

**Definition 3.10** (SAS$^+$ task)**.** *A SAS$^+$ task $\Pi^+$ is a tuple $\langle V^+, A^+, I^+, G^+\rangle$, where*

- *$V^+$ is a finite set of multi-valued variables with finite domain $dom(v)$ for each $v \in V^+$,*
- *$A^+$ is a set of actions $a = \langle pre(a), eff(a)\rangle$ where $pre(a)$ and $eff(a)$ are partial states,*
- *$I^+$ is a state and*
- *$G^+$ is a partial state.*

A state $s$ is a goal state if it is consistent with the goal description, i.e. $s(v) = G^+(v)$ for all $v \in vars(G^+)$. Action application is defined as follows:

**Definition 3.11** (SAS$^+$ action application)**.** *Given a SAS$^+$ task $\Pi^+$ with action $a$ and state $s$, $a$ is applicable in $s$ if $s(v) = pre(a)(v)$ for all $v \in vars(pre(a))$. The successor state is defined as follows:*

$$s[a](v) = \begin{cases} eff(a)(v) & \text{if } v \in vars(eff(a)) \\ s(v) & \text{otherwise} \end{cases}$$

Based on these definitions, the notions of $s$-plans and plans are defined as in the STRIPS formalism.

In order to also cover techniques using the SAS$^+$ formalism we use a compilation to STRIPS planning tasks, which has the property that both tasks have an isomorphic reachable state space. The core idea is to use one STRIPS variable for each SAS$^+$ fact.

**Definition 3.12** (SAS$^+$ to STRIPS compilation)**.** *A SAS$^+$ task $\Pi^+ = \langle V^+, A^+, I^+, G^+\rangle$ induces a STRIPS planning task $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi\rangle$ as follows:*

- *$V^\Pi = \{v_{w,d} \mid w \in V^+, d \in dom(w)\}$,*
- *each SAS$^+$ action $a$ induces a STRIPS action $a'$ with*
  - *$pre(a') = \{v_{w,d} \mid w \in V^+, d \in dom(w), pre(a)(w) = d\}$,*
  - *$add(a') = \{v_{w,d} \mid w \in V^+, d \in dom(w), eff(a)(w) = d\}$, and*
  - *$del(a') = \{v_{w,d} \mid w \in V^+, d, d' \in dom(w), eff(a)(w) = d' \neq d\}$.*

  *$A^\Pi$ is the set of all STRIPS actions induced by an SAS$^+$ action from $A^+$.*
- *$I^\Pi = \{v_{w,d} \mid w \in V^+, d \in dom(w), I^+(w) = d\}$*
- *$G^\Pi = \{v_{w,d} \mid w \in V^+, d \in dom(w), G^+(w) = d\}$*

Every state $s$ of the SAS$^+$ task corresponds to a STRIPS state $s' = \{v_{w,d} \mid s(w) = d\}$. A SAS$^+$ action $a$ is applicable in $s$ iff the induced action $a'$ is applicable in $s'$. Moreover, the successor SAS$^+$ state $s[a]$ corresponds to the successor STRIPS state $s'[a']$. One important consequence is that the reachable parts of the state space of both tasks are isomorphic, hence the SAS$^+$ task is solvable iff the STRIPS task is, and plans have a one-to-one correspondence.

For clarity, please note that there are STRIPS states that do not correspond to any SAS$^+$ state, namely all states containing $v_{w,d}$ and $v_{w,d'}$ with $d \neq d'$. However, these are *unreachable* from the initial state in the STRIPS task.

## 3.2 Representation of State Sets

Our approaches to verify unsolvability of planning tasks rely heavily on arguing over sets of states, which raises the question of how to represent these sets. Suitable representation formalisms should be able to describe sets compactly, as well as support certain query or modification operations such as "Does set $S$ contain state $s$?" or "Build the union of sets $S_1$ and $S_2$." efficiently.

Since we will often deal with large state sets we will consider formalisms based on representing state sets as logical formulas $\varphi$ over a set of variables $vars(\varphi)$. This is an idea commonly used in planning techniques dealing with large state sets, such as planning as satisfiability (Kautz and Selman, 1992) and planning as symbolic search (Edelkamp and Helmert, 2001). In this setting the STRIPS variables $V^\Pi$ are used as the variables in the formulas. The states represented by a formula over $V^\Pi$ (or a subset thereof) then correspond to the models of the formula:

**Definition 3.13** (propositional formulas representing state sets)**.** *Let $\varphi$ be a propositional logic formula over variables $V^\Pi$of a STRIPS planning task $\Pi$. The formula represents the set states($\varphi$) of states of $\Pi$ defined as:*

$$states(\varphi) = \{s_\mathcal{I} \mid \mathcal{I} : V^\Pi \to \{\top, \bot\}, \mathcal{I} \models \varphi\}, \text{ where } s_\mathcal{I} = \{v \mid \mathcal{I}(v) = \top\}.$$

For example, consider a task over variables $V = \{x, y, z\}$ and formula $\varphi = (x \vee \neg y) \wedge z)$. The states represented by $\varphi$ are $\{z\}$, $\{x, z\}$ and $\{x, y, z\}$. Variables $v \in V$ not occurring in $\varphi$ can be assigned either $\top$ or $\bot$. For example, if $V = \{x, y, z\}$ and $\varphi = x \wedge y$, then the states represented by $\varphi$ are $\{x, y\}$ and $\{x, y, z\}$.

Given two state sets $S$ and $S'$ represented by formulas $\varphi$ and $\psi$, $S \subseteq S'$ holds iff $\varphi \models \psi$ does.

In order to verify our witnesses, we will often need to express the progression or regression of a state set. Traditionally, symbolic search and planning as satisfiability introduce new variables $V^{\Pi'} = \{v' | v \in V^\Pi\}$ and define a *transition relation* $\tau_a$ over $V^\Pi \cup V^{\Pi'}$ for each action $a$:

$$\tau_a = \bigwedge_{v_p \in \mathit{pre}(a)} v_p \wedge \bigwedge_{v_a \in \mathit{add}(a)} v'_a \wedge \bigwedge_{v_d \in (\mathit{del}(a) \setminus \mathit{add}(a))} \neg v'_d \wedge \bigwedge_{v \in (V^\Pi \setminus (\mathit{add}(a) \cup \mathit{del}(a))} (v \leftrightarrow v')$$

Building the conjunction of $\tau_a$ with a formula representing a state set $S$ results in a formula that represents *pairs of states*: Non-primed variables $v \in V^\Pi$ encode those states from $S$ that meet the preconditions of $a$, and primed variables $v' \in V^{\Pi'}$ encode the successors of these states with respect to $a$.

In order to obtain a formula that only encodes the successors, one needs to apply a *forget* operation:

**Definition 3.14** (forget, Darwiche and Marquis, 2002)**.** *Let $\varphi$ be a propositional formula, and let $X$ be a set of variables. The forgetting of $X$ from $\varphi$, denoted $\exists X.\varphi$, is a formula that does not mention any variable from $X$ and for every formula $\psi'$ that does not mention any variable from $X$, we have $\varphi \models \psi$ precisely when $\exists X.\varphi \models \psi$.*

In a final step, we *rename* the primed variables back to their unprimed version, i.e. we replace each occurrence of $v' \in V^{\Pi'}$ with $v \in V^\Pi$ (denoted as $\varphi[V^{\Pi'} \rightarrow V^\Pi]$).

In summary, given formula $\varphi$ with *states*$(\varphi) = S$ and action $a$, $S[a]$ is expressed with the following formula:

$$\big(\exists V.(\varphi \wedge \tau_a)\big)[V^{\Pi'} \rightarrow V^\Pi] \tag{3.1}$$

Torralba (2015) has shown that the computation of the successor set can be performed without the need for bi-implications or renaming. The main idea is to only forget variables that are changed by the action, since the other variables do not change (which is what the bi-implications encode). By applying the forget operator on the conjunction of $\varphi$ and only the preconditions of $a$, and building the conjunction with the effects afterwards, no auxiliary variables and thus no renaming is needed:

**Definition 3.15** (progression as a formula)**.** *Given a formula $\varphi$ representing a state set $S$ (i.e. $S =$ states$(\varphi)$ and action $a$, the progression of $S$ with $a$ ($S[a]$) is represented by*

$$\Big(\exists(\mathit{add}(a) \cup \mathit{del}(a)).(\varphi \wedge \bigwedge_{v_p \in \mathit{pre}(a)} v_p)\Big) \wedge \bigwedge_{v_a \in \mathit{add}(a)} v_a \wedge \bigwedge_{v_d \in (\mathit{del}(a) \setminus \mathit{add}(a))} \neg v_d$$

For regression, we need to perform the steps in opposite order. We first built the conjunction of $\varphi$ and the add- and delete-effects. We then forget the effects and build the conjunction of the obtained formula and the preconditions:

**Definition 3.16** (regression as a formula)**.** *Given a formula $\varphi$ representing a state set $S$ (i.e. $S =$ states$(\varphi)$ and action $a$, the regression of $S$ with $a$ ($[a]S$) is represented by*

$$\Big(\exists(\mathit{add}(a) \cup \mathit{del}(a)).(\varphi \wedge \bigwedge_{v_a \in \mathit{add}(a)} v_a \wedge \bigwedge_{v_d \in (\mathit{del}(a) \setminus \mathit{add}(a))} \neg v_d)\Big) \wedge \bigwedge_{v_p \in \mathit{pre}(a)} v_p$$

### 3.2.1 Operations

A variety of formalisms based on logical formulas exist, such as CNF formulas and BDDs. Instead of analyzing for each formalism separately if they support efficient generation and verification of our witnesses, we consider an abstract formalism $\mathbf{R}$ and denote a formula represented in $\mathbf{R}$ as a $\mathbf{R}$-formula. We say that $\mathbf{R}$ is suitable for a witness if it supports certain operations efficiently. For example, any witness where we need to verify if a certain state is contained in a set can only work with formalisms that can efficiently determine if a given interpretation is a model of the formula.

In what follows we describe the set of operations we consider.[1] An $\mathbf{R}$-formula $\varphi$ is a particular instance of formalism $\mathbf{R}$. It is associated with a set of variables $\mathit{vars}(\varphi)$, which is a superset of (but not necessarily identical to) the set of variables occurring in $\varphi$. Furthermore, $\mathit{vars}(\varphi)$ follows a strict total order $\prec$. We denote the size of the representation as $\|\varphi\|$ and the amount of models as $|\varphi|$.

**MO (model testing)**
Given $\mathbf{R}$-formula $\varphi$ and truth assignment $\mathcal{I}$, test whether $\mathcal{I} \models \varphi$. Note that $\mathcal{I}$ must assign a value to all $v \in \mathit{vars}(\varphi)$ (if it assigns values to other variables not occurring in $\varphi$, they may be ignored).

**CO (consistency)**
Given $\mathbf{R}$-formula $\varphi$, test whether $\varphi$ is satisfiable.

**VA (validity)**
Given $\mathbf{R}$-formula $\varphi$, test whether $\varphi$ is valid.

**CE (clausal entailment)**
Given $\mathbf{R}$-formula $\varphi$ and clause (i.e. disjunction of literals) $\gamma$, test whether $\varphi \models \gamma$.

**IM (implicant)**
Given $\mathbf{R}$-formula $\varphi$ and cube (i.e. conjunction of literals) $\delta$, test whether $\delta \models \varphi$.

**SE (sentential entailment)**
Given $\mathbf{R}$-formulas $\varphi$ and $\psi$, test whether $\varphi \models \psi$.

**ME (model enumeration)**
Given $\mathbf{R}$-formula $\varphi$, enumerate all models of $\varphi$ (over $\mathit{vars}(\varphi)$)

**$\wedge$BC (bounded conjunction)**
Given $\mathbf{R}$-formulas $\varphi$ and $\psi$, construct an $\mathbf{R}$-formula representing $\varphi \wedge \psi$.

**$\wedge$C (general conjunction)**
Given $\mathbf{R}$-formulas $\varphi_1, \dots, \varphi_n$, construct an $\mathbf{R}$-formula representing $\varphi_1 \wedge \dots \wedge \varphi_n$.

[1] We do not consider the forget operator used in progression and regression since we can express all needed queries without it, as we will see later.

**$\vee$BC (bounded disjunction)**
Given $\mathbf{R}$-formulas $\varphi$ and $\psi$, construct an $\mathbf{R}$-formula representing $\varphi \vee \psi$.

**$\vee$C (general disjunction)**
Given $\mathbf{R}$-formulas $\varphi_1, \ldots, \varphi_n$, construct an $\mathbf{R}$-formula representing $\varphi_1 \vee \cdots \vee \varphi_n$.

**$\neg$C (negation)**
Given $\mathbf{R}$-formula $\varphi$, construct an $\mathbf{R}$-formula representing $\neg\varphi$.

**CL (conjunction of literals)**
Given a conjunction $\varphi$ of literals, construct an $\mathbf{R}$-formula representing $\varphi$.

**RN (renaming)**
Given $\mathbf{R}$-formula $\varphi$ and an injective variable renaming $r : \mathit{vars}(\varphi) \to V'$, construct an $\mathbf{R}$-formula representing $\varphi[r]$, i.e., $\varphi$ with each variable $v$ replaced by $r(v)$.

**RN$_\prec$ (renaming consistent with order)**
Same as **RN**, but $r$ must be consistent with the variable order in the sense that if $v_1, v_2 \in \mathit{vars}(\varphi)$ with $v_1 \prec v_2$, then $r(v_1) \prec r(v_2)$.

**toCNF (transform to CNF)**
Given $\mathbf{R}$-formula $\varphi$, construct a CNF formula that is equivalent to $\varphi$.

**toDNF (transform to DNF)**
Given $\mathbf{R}$-formula $\varphi$, construct a DNF formula that is equivalent to $\varphi$.

**CT (model count)**
Given $\mathbf{R}$-formula $\varphi$, count how many models $\varphi$ has.

We say an $\mathbf{R}$-formula *efficiently* supports an operation if it can perform it in time polynomial in the size of the involved $\mathbf{R}$-formula(s) *except for the case of **ME***. For **ME** we only require that the operation can be performed in time polynomial in the size of $\varphi$ *and the amount of models of* $\varphi$ (over $\mathit{vars}(\varphi)$). Since we still want to guarantee that the witness can be generated with polynomial overhead and verified in polynomial time in its size, we need to pay special attention whenever we use **ME** and ensure that the input is polynomial in the amount of models.

### 3.2.2 Specific Formalisms

While we will consider general $\mathbf{R}$-formalisms when discussing our results, we also want to focus on some concrete formalisms which are oftentimes suitable and also serve as formalisms for concrete implementations of our verification systems. Table 3.1 shows an overview which operations are supported efficiently by which formalism. Unless stated otherwise, these results originate from Darwiche and Marquis (2002). For clarity we will however still give an brief reasoning for the results.

| | BDD | Horn | 2CNF | MODS |
|---|---|---|---|---|
| **MO** | yes | yes | yes | yes |
| **CO** | yes | yes | yes | yes |
| **VA** | yes | yes | yes | yes |
| **CE** | yes | yes | yes | yes |
| **IM** | yes | yes | yes | yes |
| **SE** | yes | yes | yes | yes |
| **ME** | yes | yes | yes | yes |
| $\wedge$**BC** | yes | yes | yes | yes |
| $\wedge$**C** | no | yes | yes | no* |
| $\vee$**BC** | yes | no | no | no* |
| $\vee$**C** | no | no | no | no* |
| $\neg$**C** | yes | no | no | no |
| **CL** | yes | yes | yes | yes |
| **RN** | no | yes | yes | yes |
| $\mathbf{RN}_{\prec}$ | yes | yes | yes | yes |
| **toDNF** | no | no | no | yes |
| **toCNF** | no | yes | yes | yes |
| **CT** | yes | (no) | (no) | yes |

* MODS supports $\wedge$**C**, $\vee$**BC** and $\vee$**C** efficiently if all involved formulas are over the same set of variables.

Table 3.1: Comparison of efficient operations support for different formalisms. Entry "(no)" means that the operation is not efficiently supported unless **P** = **NP**.

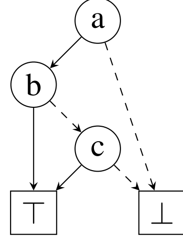



Figure 3.2: A BDD representing formula $a \wedge (b \vee c)$. A solid outgoing edge from a node with variable $v$ means that the truth assignment for $v$ is $\top$, a dashed edge that it is $\bot$.

### BDDs

Reduced Ordered Binary Decision Diagrams (Bryant, 1986) or ROBDDs represent logical formulas as a directed acyclic graph. Besides the two terminal nodes $\top$ and $\bot$, all nodes $n$ are associated with a variable $v_n$ and have exactly two outgoing edges, which represent assigning $\top$ or $\bot$ to $v$. ROBDDs follow a *variable ordering* $\prec$ in the sense that for each edge $\langle n, n' \rangle$ we have $v_n \prec v_{n'}$. They are *reduced* because nodes that are redundant (both outgoing edges lead to the same node) or duplicate (another node has the same true and false children) are removed. For two equivalent formulas and a given variable order, their ROBDD representation is isomorphic. In literature the term BDDs usually refers to ROBDDs, and we adopt this notion in what follows.

Figure 3.2 shows a BDD for the formula $a \wedge (b \vee c)$ with variable ordering $a \prec b \prec c$. All paths from the root node to $\top$ represent models of the formula. If a variable does not occur on a path, then it can be assigned either value. For example the path following the solid edges from $a$ and $b$ does not contain a node for $c$, thus it represents both assignments $\{a \mapsto \top, b \mapsto \top, c \mapsto \top\}$ and $\{a \mapsto \top, b \mapsto \top, c \mapsto \bot\}$.

BDDs efficiently support **MO**, **CO**, **VA**, **CE**, **IM**, **SE**, **ME**, $\wedge$**BC**, $\vee$**BC**, $\neg$**C**, **CL**, **RN**$_\prec$ and **CT**:

- **MO**: Follow the path corresponding to the given truth assignment and see if it leads to $\top$, which can be done in time linear in $|V|$.
- **CO**, **VA**: Any inconsistent formula is represented by a BDD consisting of only $\bot$ as root. If the given BDD does not have $\bot$ as root, it is consistent, which we can check in constant time. The same argument works for **VA** with $\top$ as root.
- **ME**: Bryant (1986) presents an algorithm that given a BDD $B$ representing a formula $\varphi$ enumerates all models of $\varphi$ in time $O(n * |\varphi|)$, where $n$ is the amount of variables in $B$ and $|\varphi|$ the amount of models of $\varphi$.
- $\wedge$**BC**, $\wedge$**C**, $\vee$**BC**, $\vee$**C**: Building the conjunction or disjunction of two BDDs $B$ and $B'$ is a special case of the *Apply* operation which has complexity $\|B\| \cdot \|B'\|$ (Bryant, 1986). Since the size of the resulting BDD is $\|B\| \cdot \|B'\|$ in worst case as well,

applying $n$ conjunctions or disjunctions is exponential in $n$, thus it is polynomial for bound $n$ but exponential in the general case.

- $\neg$**C**: Given a BDD $B$ for some formula $\varphi$, the BDD for $\neg\varphi$ is equivalent to $B$ except that $\top$ and $\bot$ are switched, and can be built in time linear in $\|B\|$. However, modern BDD implementations optimize negation by adding *complement edges* (Brace, Rudell, and Bryant, 1990), making the operation constant.

- **CL**: If the conjunction is inconsistent (i.e. a variable occurs both positive and negative) the BDD consists only of the $\bot$ leaf. Otherwise we can build the BDD in time linear in the size of the conjunction, independent of variable ordering: Each variable is represented by one node, with children $\bot$ and the node of the next ordered variable occurring in the conjunction (or $\top$ if it is the last variable). If the literal is positive in the conjunction, then the false-edge leads to $\bot$, otherwise the true-edge.

- **CE**, **IM**, **SE**: Testing whether $\varphi \models \psi$ is equivalent to asking whether $\varphi \wedge \neg\psi$ is inconsistent. Building a BDD representing $\varphi \wedge \neg\psi$ can be done in polynomial time with $\neg$**C** and $\wedge$**BC**, and testing consistency of the result is also polynomial with **CO**. In case of **CE** we have $\varphi \models \gamma$ for a clause $\gamma$, meaning we need to build a BDD for $\gamma$ first, which we can do similar to **CL** in time linear in $|\gamma|$. In case of **IM**, we have $\delta \models \varphi$, where $\delta$ is a conjunction of literals and can thus be built efficiently.

- **RN**, **RN**$_{\prec}$: Renaming for BDDs amounts to changing the variable order. Since different variable orders for the same formula can cause an exponential difference in size (Bryant, 1986), general renaming cannot be performed efficiently. If however the renaming is consistent with the variable order, the graph structure of the BDD does not change and such a BDD can thus be built in time linear in the size of the original BDD.

- **toDNF**, **toCNF**: The odd-parity function of $n$ variables can be represented by a BDD with $2n + 1$ nodes, but any CNF or DNF representing this function contains $O(2^n)$ clauses or cubes (Håstad, 1987). Thus a BDD representing this function cannot efficiently be translated into CNF or DNF.

- **CT**: Bryant (1986) shows that counting can be done in time linear in the number of nodes of the BDD.

### Horn and 2CNF Formulas

Horn and 2CNF formulas are special cases of CNF formulas, where in a Horn formula each clause contains at most one positive literal (and an arbitrary amount of negative literals), and in a 2CNF formula each clause contains at most two literals. They are two of the maximal tractable classes in Schaefer's dichotomy (Schaefer, 1978) for Boolean

constraint satisfaction, meaning we can decide the satisfiability problem for them in polynomial time.

The key idea of checking consistency for Horn formulas is to iteratively select a clause consisting of only one positive literal, set this variable to true and propagate the assignment to all other clauses. If at some point a clause becomes empty, the formula is unsatisfiable. If no more unit clauses with one positive literal exist, setting all unassigned variables to false will result in a model of the formula.

For 2CNF formulas, several polytime algorithms for deciding satisfiability are known. Aspvall, Plass, and Tarjan (1979) describe a linear time algorithm where we first build an implication graph from the formula with nodes $v$ and $\neg v$ for each variable $v$ and edges from $\neg l$ to $l'$ and $l$ to $\neg l'$ for each clause $l \vee l'$. Next, we find all strongly connected components. If any component contains $v$ as well as $\neg v$ for some variable $v$, the formula is unsatisfiable. If this is not the case, one can find an assignment by merging the nodes of each strongly connected component, order the nodes topologically and then iteratively mark the lowest-ordered unmarked node and its literals with $\top$ and the node representing the complement of the marked literals with $\bot$.

Not all propositional formulas have equivalent Horn or 2CNF formulas. To see this, we first establish that since any kind of CNF formula is a conjunction of clauses it must imply each of its clauses. For Horn formulas, consider now the formula $a \vee b$. It is not a Horn formula, and any equivalent Horn formula $\varphi$ must consist of only clauses that are implied by $a \vee b$. The only such clauses are either equivalent to true (if they contain a variable and its negation) or must contain $a$ and $b$ and can thus not be a part of $\varphi$, meaning $\varphi$ can only be a conjunction of clauses equivalent to $\top$. Thus $\varphi$ itself is equivalent to $\top$ and not to $a \vee b$. A similar argument can be made for formula $a \vee b \vee c$ and 2CNF, since the only clauses implied by $a \vee b \vee c$ either contain more than two literals or are equivalent to $\top$.

Horn and 2CNF formulas efficiently support **MO**, **CO**, **VA**, **CE**, **IM**, **SE**, **ME**, $\wedge$**BC**, $\wedge$**C**, **CL**, **RN**, **RN**$_\prec$ and **toCNF**:

- **MO**: As with any propositional formula, evaluating a given assignment can be done in linear time.

- **CO**: Consistency can be checked with the algorithms described above.

- **VA**: We can check if a CNF formula is valid by checking if each clause is valid, which is the case iff for some variable $v$ both $v$ and $\neg v$ appear in it.

- **CE**, **IM**, **SE**: $\varphi \models \psi$ is equivalent to testing if $\varphi \wedge \neg\psi$ is inconsistent. If $\psi$ is a clause, its negation is a set of unit clauses and thus $\varphi \wedge \neg\psi$ is a Horn/2CNF formula if $\varphi$ is, meaning we can efficiently check consistency. If $\psi = \bigwedge c_i$ is a Horn formula, then $\varphi \wedge \neg\psi \equiv \varphi \wedge \bigvee \neg c_i$ is inconsistent iff $\varphi \wedge \neg c_i$ is inconsistent for each $c_i$. Since $c_i$ is a clause, this reduces to a number of **CE** checks. The same arguments can be made for $\varphi$ being 2CNF. **IM** is also covered through **SE**, since $\delta$ is a conjunction of literals and thus both a Horn and 2CNF formula

- **ME**: Algorithms for efficient enumeration for both Horn and 2CNF formulas can be found in Dechter and Itai (1992).
- $\wedge$**BC**, $\wedge$**C**: In order to conjoin two Horn/2CNF formulas, we simply need to merge the two clause sets.
- $\vee$**BC**, $\vee$**C**, $\neg$**C**: The disjunction of two Horn/2CNF formulas or the negation of one is in general not a Horn/2CNF formula, and as shown above for some formulas it is impossible to rephrase it into one. Thus $\vee$**BC**, $\vee$**C** and $\neg$**C** are not supported (not even inefficiently).
- **CL**: Each literal by itself is a clause with at most one positive variable, thus their conjunction is both a Horn and 2CNF formula.
- **RN**, **RN**$_\prec$: Renaming can be performed in time linear in the size of the formula by simply iterating over the formula and replacing $V$ with $r(V)$.
- **toDNF**, **toCNF**: Horn and 2CNF formulas are CNF formulas by design, but transforming for example $\bigwedge(\neg x_i \vee \neg y_i)$ (which is both Horn and 2CNF) to DNF causes exponential blowup (Miltersen, Radhakrishnan, and Wegener, 2005).
- **CT**: Counting the number of models is **#P**-complete for both Horn (Hermann and Pichler, 2010) and 2CNF (Valiant, 1979), and thus no efficient algorithm exists unless **P** $=$ **NP**.

**Explicit Enumeration (MODS)**

We consider explicit enumeration as an enumeration of models over a set of variables. It can be seen as a DNF $\varphi$ where each cube involves all variables $vars(\varphi)$, is consistent and is disjoint from all other cubes (i.e. $c_i \wedge c_j \equiv \bot$ for all cubes $c_i, c_j$ in $\varphi$). It is the only considered formalism where the representation size $\|\varphi\|$ is linear in the amount of models $|\varphi|$.

MODS efficiently supports **MO**, **CO**, **VA**, **CE**, **IM**, **SE**, **ME**, $\wedge$**BC**, **CL**, **RN**, **RN**$_\prec$, **toDNF**, **toCNF** and **CT**, as well as $\wedge$**C**, $\vee$**BC** and $\vee$**C** if all involved formulas are over the same set of variables:

- **MO**: Evaluating a given assignment can be done in linear time by iterating over all models and comparing them to the sought model.
- **CO**: As long as $\varphi$ contains at least one cube the formula is satisfiable.
- **VA**: A MODS formula is valid if it contains $2^{vars(\varphi)}$ cubes, and the number of cubes can be checked in time linear in the formula size.
- **CE**: We can check for each cube $c_i$ in $\varphi$ that it implies the given clause $\gamma$, which is the case if at least one literal $l$ appears in both $c_i$ and $\gamma$.

- **IM**, **SE**: We first consider **SE**. If *vars*$(\varphi) =$ *vars*$(\psi)$, we can enumerate each model of $\varphi$ with **ME** and check if it is a model of $\psi$ with **MO** (we can use **ME** without introducing exponential blowup since $|\varphi|$ is linear in $\|\varphi\|$). In the general case we can expand each model $m$ of $\varphi$ to $2^{(vars(\psi) \setminus vars(\varphi))}$ models $m_i$ over *vars*$(\varphi) \cup$ *vars*$(\psi)$ (covering all possible assignments to (*vars*$(\psi) \setminus$ *vars*$(\varphi)$) and then restrict each $m_i$ to *vars*$(\psi)$, resulting in $m'_i$. Enumerating $m'_i$ incrementally avoids an exponential blowup. Given $m$, each $m'_i$ is distinct, and since $\psi$ has $|\psi|$ distinct models we will find a $m'_i$ that is not a model of $\psi$ at the latest in step $|\psi| + 1$ (if there are that many $m'_i$) and can terminate the query. This results in overall complexity $O(\|\varphi\| \cdot \|\psi\|)$. **IM** is covered by this as well, since $\delta$ is a conjunction of literals and thus a MODS formula.

- **ME**: Each cube corresponds to exactly one model, thus we can just enumerate all cubes.

- $\wedge$**BC**, $\wedge$**C**: $\varphi \wedge \psi$ is an enumeration of models over *vars*$(\varphi) \cup$ *vars*$(\psi)$ and can be calculated by building the conjunction $m_i \wedge m_j$ for all models $m_i$ of $\varphi$ and all models $m_j$ of $\psi$ (duplicate literals are eliminated and if the resulting model is inconsistent it is ignored), which is possible in time $O(\|\varphi\| \cdot \|\psi\|)$. For unbounded conjunction, consider $\varphi = \bigcap_{i=1}^{n}(x_{i,1} \vee x_{i,2})$. Each clause $(x_{i,1} \vee x_{i,2})$ has 3 models (over *vars*$(x_{i,1} \vee x_{i,2})$), but $\varphi$ has $3^n$ models (over *vars*$(\varphi)$) and thus cannot be built efficiently. If however all subformulas in the conjunction are over the same variables, we can iterate over each model in the first subformula, check if it is contained in all other subformulas and if so add it to the result.

- $\vee$**BC**, $\vee$**C**: Disjunction cannot be efficiently supported unless *vars*$(\varphi) =$ *vars*$(\psi)$: Assume *vars*$(\varphi) = \{x_1\}$ and *vars*$(\psi) = \{x_1, \ldots, x_n\}$, and both $\varphi$ and $\psi$ have only one model. An enumeration of the disjunction must be over *vars*$(\psi)$, meaning we need to extend the one model from $\varphi$ over $\{x_1\}$ to $2^{n-1}$ models over $\{x_1, \ldots x_n\}$. However, if *vars*$(\varphi) =$ *vars*$(\psi) = V$, then the disjunction is also over $V$ and we can simply merge the cubes (while eliminating possible duplicates).

- $\neg$**C**: Consider a MODS formula over $n$ variables with only one model. The negation of this formula must contain $2^n - 1$ models, leading to an exponential blowup.

- **CL**: A conjunction of literals can be seen as a MODS formula over variables occurring in the conjunction with only one model if the conjunction is consistent, otherwise it can be represented by the constant $\perp$ formula.

- **RN**, **RN**$_\prec$: Renaming can be performed in the same way as for Horn and 2CNF by simply iterating over all cubes and replacing each occurrence of $v \in V$ with $r(v)$.

- **toDNF**, **toCNF**: Since a MODS-formula is already a DNF, nothing needs to be done for **toDNF**. For **toCNF**, we first incrementally build a decision tree from the

models. We then form the negation of each path from the root of the tree to $\bot$, the conjunction of these clauses forms a CNF representing the same formula (Darwiche and Marquis, 2002).

- **CT**: Since MODS is an enumeration of models we can count the amount of models in time linear in $\|\varphi\|$ by simply iterating over $\varphi$.

# Part I

# Witnesses

# 4 Inductive Certificates

The first type of witness we propose is based on *inductive sets*, which informally speaking are sets that once entered cannot be left again. In Section 4.1 we formally define and investigate properties of inductive sets. Section 4.2 introduces and analyses a first inductive certificate variant that argues with a single inductive set. However, this approach is quite limited since it might not always be possible to describe a single set efficiently. In order to extend the applicability of the concept of inductive certificates, Sections 4.3 and 4.4 introduce two variants where the inductive set is represented by a union or intersection of a family of sets.

## 4.1 Inductive Sets

Proving that a planning task is unsolvable essentially requires showing that in the graph induced by the planning task (i.e. the search space) there is no path from the initial state to any state satisfying the goal description. Proving this path-nonexistence is easier if the graph is acyclic. In this case, all paths from the initial state must end in a non-goal state with no successor. We can first directly argue that no $s$-plan can exist for these states since they are not goal states and no action is applicable; and then iteratively argue for their predecessors that no $s$-plan can exist since they are not goal states and for all their successors $s[a]$ no $s[a]$-plan exists.

However, graphs induced from planning problems are in general not acyclic. Thus we might lack the first step of the above recursive proof idea. For example, consider a planning task consisting of variables $V = \{a, b, g\}$, initial state $I = \{a\}$, goal $G = \{g\}$ and two actions: $a_1$ can be applied if $a$ is true, deletes $a$ and adds $b$, and $a_2$ can be applied if $b$ is true, deletes $b$ and adds $a$. The task is obviously unsolvable since the initial state does not contain $g$ and no action can add $g$. While we can only reach two states from $I$ ($I = \{a\}$ itself and $\{b\}$), both these states have successors, making the above proof idea inapplicable.

Instead we need to argue with *sets of states*. Analogously to a state with no successors we can define a set with no successors in the sense that any successor of a state in the set is contained within the same set.

**Definition 4.1** (inductive set)**.** *A set $S \subseteq S^{\Pi}$ of states of a STRIPS planning task $\Pi = \langle V^{\Pi}, A^{\Pi}, I^{\Pi}, G^{\Pi} \rangle$ is* forward inductive *(or simply* inductive*) in $\Pi$ if $S[A^{\Pi}] \subseteq S$, i.e., all action applications to a state in $S$ lead to a state that is also in $S$.*

*If $\Pi$ is clear from context, we simply say $S$ is inductive.*

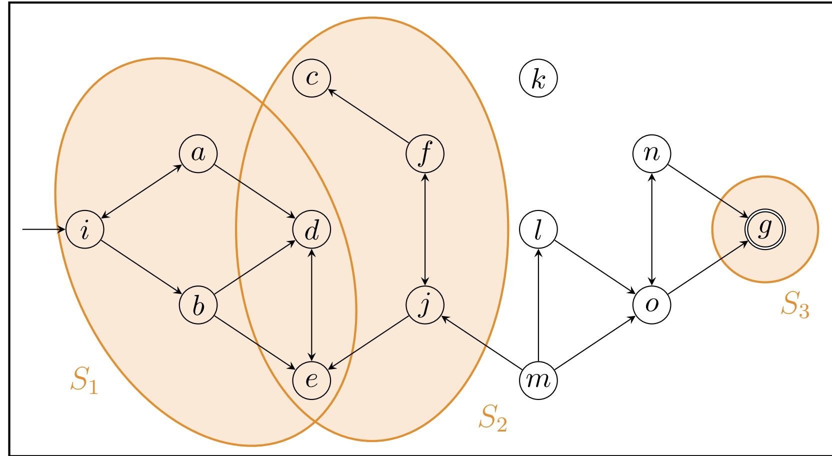



Figure 4.1: Examples of inductive sets in a search space.

Figure 4.1 shows some examples of inductive sets in the search space of a planning task. As we see from $S_3$, states with no successors are a special case of inductive sets where the set contains exactly one element.

Inductive sets do not correspond to connected components as we can show with the following examples from Figure 4.1:

- $S = \{m\}$ is strongly connected (and thus also weakly connected) since it consists of only one state, but it is not inductive since $m$ has transitions leading outside of $S$.
- $S' = \{k, g\}$ is not weakly connected (and thus also not strongly connected) but inductive since neither $k$ nor $g$ have outgoing transitions.

They are however connected to strongly connected components (SCC) in the sense that an inductive set is always a collection of SCCs, which also means that an inductive set can never contain only part of an SCC. Furthermore, if we merge all states in the same SCC to one node (resulting in a directed *acyclic* graph), an inductive set must contain all the descendants of each node whose states are contained.

Inductive sets strongly relate to reachability. Consider an arbitrary state $s$ and set $\mathcal{R}(s)$ consisting of exactly the states reachable from $s$ (including $s$ itself). There can be no edge from any $s' \in \mathcal{R}(s)$ to some $s'' \notin \mathcal{R}(s)$ since otherwise $s''$ would also be reachable from $s$, leading to the following proposition:

**Proposition 4.1.** *Given state $s$, the set $\mathcal{R}(s)$ of states reachable from $s$ is inductive.*

We can also say that if an inductive set contains a state $s$, it must contain all of $\mathcal{R}(s)$. From this follows directly that an inductive set is the union of all $\mathcal{R}(s)$ for all states $s$ in $S$:

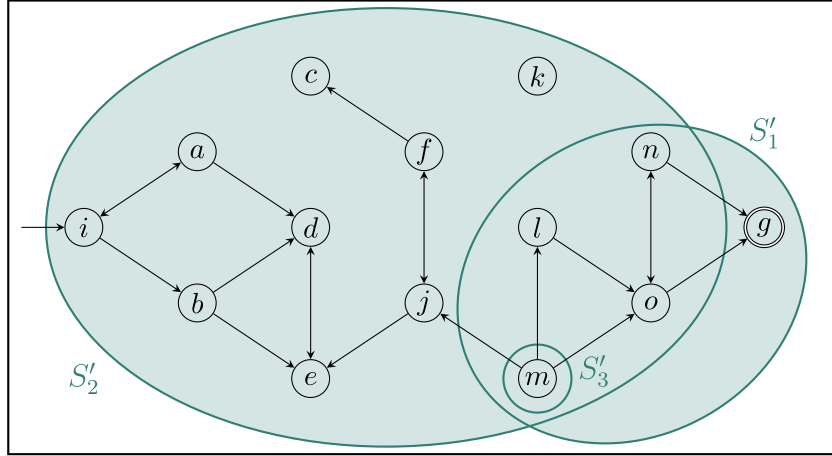



Figure 4.2: Examples of backwards inductive sets in a search space.

**Proposition 4.2.** *If $S$ is inductive, then $S = \bigcup_{s \in S} \mathcal{R}(s)$.*

Another interesting property of inductive sets is that for two inductive sets their union and their intersection is inductive as well, as we can see for $S_1$ and $S_2$ in Figure 4.1.

**Theorem 4.1** (closure properties of inductive sets)**.** *Given inductive sets $S$ and $S'$:*

*(1) $S \cup S'$ is inductive.*

*(2) $S \cap S'$ is inductive.*

**Proof:**

(1) Consider any arbitrary $s \in S \cup S'$ and an arbitrary action $a$ applicable to $s$. If $s \in S$, then $s[a] \in S$ must be true since $S$ is inductive, from which follows that $s[a] \in S \cup S'$. If $s \notin S$, then $s \in S'$ and $s[a] \in S \cup S'$ with the same argument (since $S'$ is also inductive). Since we chose $s$ and $a$ arbitrarily, we have shown that all successors of all states in $S \cup S'$ are also in $S \cup S'$, thus $S \cup S'$ is inductive.

(2) Consider any arbitrary $s \in S \cap S'$ and $a$ applicable to $s$. From $s \in S \cap S'$ we have $s \in S$ and $s \in S'$. Since both $S$ and $S'$ are inductive, we know that $s[a] \in S$ and $s[a] \in S'$, which means $s \in S \cap S'$. From this we conclude that $S \cap S'$ is inductive.

□

### 4.1.1 Backwards Inductive Sets

The concept of inductive sets can also be applied in a regression perspective, i.e. a set that cannot be entered from the outside:

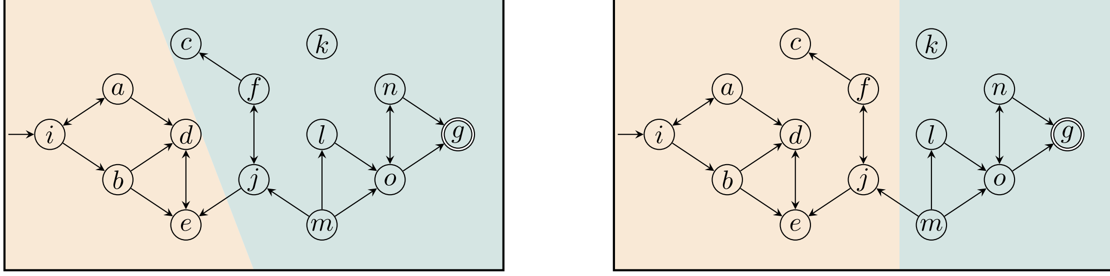



Figure 4.3: Two examples showing how the complement of an inductive set (in orange) is backwards inductive (in turquoise).

**Definition 4.2** (backwards inductive set)**.** *A state set $S$ is* backwards inductive *iff for all $s \in S$ and $a$ backwards-applicable in $s$ we have $[a]s \in S$.*

Figure 4.2 shows some examples of backwards inductive sets in the same search space as Figure 4.1. Similar to forward inductive sets we see that states with no predecessors are a special case of backwards inductive sets.

If we look at $S_2'$, we see that it contains exactly the states not contained in $S_3$ from Figure 4.1. This is not by chance, as inductivity and backwards inductivity are closely related:

**Theorem 4.2.** *$S$ is inductive iff $\overline{S}$ is backwards inductive.*

**Proof:** If $S$ is inductive, then we have $s[a] \in S$ for any $s \in S$ and $a \in A^\Pi$ applicable to $s$. This means that for any $s' \notin S$ and backwards-applicable action $a'$ the predecessor $s'[a']$ cannot be in $S$ thus must be in $\overline{S}$, meaning $\overline{S}$ is backwards inductive. For the opposite direction we have that all predecessors of $\overline{S}$ are in $\overline{S}$, meaning there can be no state $s \in \overline{\overline{S}} = S$ with $s[a] \in \overline{S}$, thus $S$ must be inductive. □

This means we can consider an inductive or backwards inductive set as partitioning the search space in two areas, where we cannot reach one from the other. Note however that this separation is not bidirectional: if $S$ is inductive, then we cannot reach $\overline{S}$ from $S$, but we might reach $S$ from $\overline{S}$, i.e. there might be a transition from $s \in \overline{S}$ to $s' \in S$. Figure 4.3 shows two examples of such a partitioning.

## 4.2 Simple Inductive Certificates

The most natural way to argue that a task $\Pi$ is unsolvable is that the set of states reachable from the initial state $I^\Pi$ does not contain any goal states. A certificate could then consist of simply this set $S$ of reachable states. In order to verify this form of certificate the verifier would need to check that $S$ does not contain any goal state and $S$ is indeed the set of states reachable from $I^\Pi$.

While checking the exclusion of all goal states from $S$ might be doable in an efficient manner, verifying that a set is equivalent to the set of states reachable from $I^\Pi$ is not straightforward. A verifier could check it by conducting a blind search from $I^\Pi$ and see if the set of states it finds is equivalent to $S$. This is however in general not an efficient verification since it requires $|S|$ (the amount of states in $S$) expansion steps, but $\|S\|$ (the representation size of $S$) might be exponentially smaller than $|S|$. Consider for example a planning task where all states are reachable from $I^\Pi$. In this case $S$ can be represented by $\top$, but exploring the search space requires $2^{V^\Pi}$ expansions.

Instead we observe that we do not need the exact set of states reachable from $I^\Pi$; we only need to know that *no goal state can be reached from* $I^\Pi$. We know that given an inductive set $S$, any state $s \in S$ cannot reach any state $s \notin S$. Thus an inductive set that contains $I^\Pi$ but does not contain any goal state shows that no goal can be reached by $I^\Pi$, meaning such an inductive certificate is a witness for unsolvable planning tasks.

**Definition 4.3** (inductive certificate)**.** *Given a task* $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$*, an* inductive certificate for $\Pi$ *is a set* $S \subseteq S^\Pi$ *of states such that*

1. $I^\Pi \in S$*,*
2. $S \cap S^\Pi_G = \emptyset$*, and*
3. $S$ *is inductive in* $\Pi$*.*

As we stated in Chapter 1, one requirement for our witnesses is that they are sound and complete. For inductive certificates, this is the case:

**Theorem 4.3** (soundness and completeness of inductive certificate)**.** *Given a STRIPS planning tasks* $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$*, there is an inductive certificate for* $\Pi$ *iff* $\Pi$ *is unsolvable.*

**Proof:**

**soundness:** If an inductive certificate for $\Pi$ exists, then it contains all states reachable from $I^\Pi$. Since it also cannot contain any goal state, it follows that no goal state is reachable from $I^\Pi$, meaning the task is unsolvable.

**completeness:** If a planning task is unsolvable, no goal state is reachable from $I^\Pi$. Together with Proposition 4.1 we conclude that the set of states reachable from $I^\Pi$ then forms an inductive certificate. □

Aside from soundness and completeness, we want our certificates to be efficiently generated, efficiently verifiable, and general in the sense that they are applicable to a variety of planning techniques. Efficient generation and generality depends on the concrete planning techniques and will thus only be addressed in part II of this thesis. Efficient verification on the other hand only depends on the representation used for $S$. An inductive certificate represented by formalism $\mathbf{R}$ is called an *inductive* $\mathbf{R}$*-certificate*.

The most difficult part of verifying an inductive certificate is to show inductivity. Definition 3.15 specifies how we can express the progression of $S$ with $a$, but uses the forget operator. In the case of verifying inductivity however, we do not need it. We first show the following lemma:

**Lemma 4.1.** *Let $\varphi$, $\psi$ and $\chi$ be propositional formulas, and $X$ and $X'$ be propositional variables, where $\varphi$, $\psi$ and $\chi$ do not mention any variable from $X'$. The statement $(\exists X.\varphi) \wedge \psi \models \chi$ holds iff $\varphi[X \to X'] \wedge \psi \models \chi$ does.*

**Proof:** We observe that $\exists X.\varphi$ is equivalent to $\exists X'.(\varphi[X \to X'])$ if $\varphi$ does not mention any variable from $X'$, and thus $(\exists X.\varphi)\wedge\psi \models \chi$ holds iff (1) $(\exists X'.(\varphi[X \to X']))\wedge\psi \models \chi$ does. According to the deduction theorem, (1) in turn holds iff (2) $\exists X'.(\varphi[X \to X']) \models \neg\psi \vee \chi$ does.

Since $\neg\psi\vee\chi$ does not mention any variable from $X'$, we can now apply Definition 3.14 and state that (2) holds iff (3) $\varphi[X \to X'] \models \neg\psi \vee \chi$ does. Applying the deduction theorem again yields that (3) holds iff $\varphi[X \to X'] \wedge \psi \models \chi$ does, concluding the proof. □

Following this lemma the next theorem states which operations $\mathbf{R}$ must efficiently support in order to efficiently verify an inductive certificate:

**Theorem 4.4.** *Inductive $\mathbf{R}$-certificates $\varphi$ for $\Pi$ can be verified in polynomial time in $\|\varphi\|$ and $\|\Pi\|$ if $\mathbf{R}$ efficiently supports **MO**, **CE**, **SE**, $\wedge$**BC**, **CL** and **RN**$_\prec$.*

**Proof:** To verify that $\varphi$ is an inductive $\mathbf{R}$-certificate for $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi\rangle$, we must verify that it satisfies the three requirements of Definition 4.3, where $S = \mathit{states}(\varphi)$.

Requirement 1 ($I^\Pi \in \mathit{states}(\varphi)$) can be verified using **MO** by testing $\mathcal{I} \models \varphi$ with the truth assignment $\mathcal{I}$ defined as $\mathcal{I}(v) = \top$ for $v \in I^\Pi$ and $\mathcal{I}(v) = \bot$ otherwise.

Requirement 2 ($\mathit{states}(\varphi) \cap S_G^\Pi = \emptyset$) can be verified as $\varphi \models \bigvee_{v\in G^\Pi} \neg v$ using **CE**.

For requirement 3 ($\mathit{states}(\varphi)$ is inductive in $\Pi$), we can verify for each action $a \in A^\Pi$ individually that $\mathit{states}(\varphi)[a] \subseteq \mathit{states}(\varphi)$. With the definition of $\mathit{states}(\varphi)[a]$ from Definition 3.15 we reformulate this statement into the following entailment:

$$\Big(\exists(\mathit{add}(a) \cup \mathit{del}(a)).(\varphi \wedge \bigwedge_{v_p\in \mathit{pre}(a)} v_p)\Big) \wedge \bigwedge_{v_a\in \mathit{add}(a)} v_a \wedge \bigwedge_{v_d\in(\mathit{del}(a)\setminus \mathit{add}(a))} \neg v_d \models \varphi \quad (4.1)$$

Applying Lemma 4.1 yields that the entailment in (4.1) holds iff

$$\Big((\varphi\wedge \bigwedge_{v_p\in \mathit{pre}(a)} v_p)[(\mathit{add}(a)\cup \mathit{del}(a)) \to X']\Big)\wedge \bigwedge_{v_a\in \mathit{add}(a)} v_a\wedge \bigwedge_{v_d\in(\mathit{del}(a)\setminus \mathit{add}(a))} \neg v_d \models \varphi \quad (4.2)$$

does, where $X'$ is a fresh set of variables. This test can be performed with **SE**, **CL**, **RN** and $\wedge$**BC**.

Instead of using an arbitrary $X'$, we introduce a fresh auxiliary propositional variable $v'$ for every state variable $v \in V^\Pi$. We write $V^{\Pi'}$ for these new ("primed") variables. We

ensure that each primed variable is adjacent to its unprimed variable in the variable order: if $v_1 \prec \cdots \prec v_n$ is the order on $V^\Pi$, then $v_1 \prec v_1' \prec \cdots \prec v_n \prec v_n'$ is the order on $V^\Pi \cup V^{\Pi'}$. The variables that have to be renamed are then renamed to their respective primed variables. This enables us to use $\mathbf{RN}_\prec$ instead of **RN**. □

From this we can conclude that BDDs, Horn formulas, 2CNF formulas and MODS all support efficient verification of inductive certificates.

A weakness of inductive certificates is that we need to describe the set as a single $\mathbf{R}$-formula. In order to cover cases where this is not possible or requires a prohibitive representation size, we investigate situations where we can compactly represent the set as a *union* or *intersection* of several $\mathbf{R}$-formulas in the next two sections.

## 4.3 Disjunctive Certificates

We first discuss disjunctive certificates, which represent an inductive certificate as the union over a family of sets:

**Definition 4.4** (disjunctive certificate). *A family $\mathcal{F} \subseteq 2^{S^\Pi}$ of state sets of task $\Pi$ is called a* disjunctive certificate *for $\Pi$ if $\bigcup_{S \in \mathcal{F}} S$ is an inductive certificate for $\Pi$.*

The goal of introducing composite certificates is to deal with sets that cannot be compactly represented in a single set. However, in order to verify that a family of sets is indeed a disjunctive certificate, we would need to explicitly build the union and test whether this union is an inductive certificate, negating the advantage of representing the set as a union of sets. In order to achieve efficient verification we define a restricted variant with stronger properties that can be checked without building the union of all sets and imply the properties of inductive sets:

**Definition 4.5** ($r$-disjunctive certificate). *For $r \in \mathbb{N}_0$, a family $\mathcal{F} \subseteq 2^{S^\Pi}$ of state sets of task $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$ is called an* $r$-disjunctive certificate *if:*

1. *$I^\Pi \in S$ for some $S \in \mathcal{F}$,*
2. *$S \cap S_G^\Pi = \emptyset$ for all $S \in \mathcal{F}$, and*
3. *for all $S \in \mathcal{F}$ and all $a \in A^\Pi$, there is a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ with $|\mathcal{F}'| \leq r$ such that $S[a] \subseteq \bigcup_{S' \in \mathcal{F}'} S'$.*

We call the third property of the definition *disjunctive $r$-inductivity*. We now establish that $r$-disjunctive certificates prove unsolvability:

**Theorem 4.5.** *All $r$-disjunctive certificates are disjunctive certificates.*

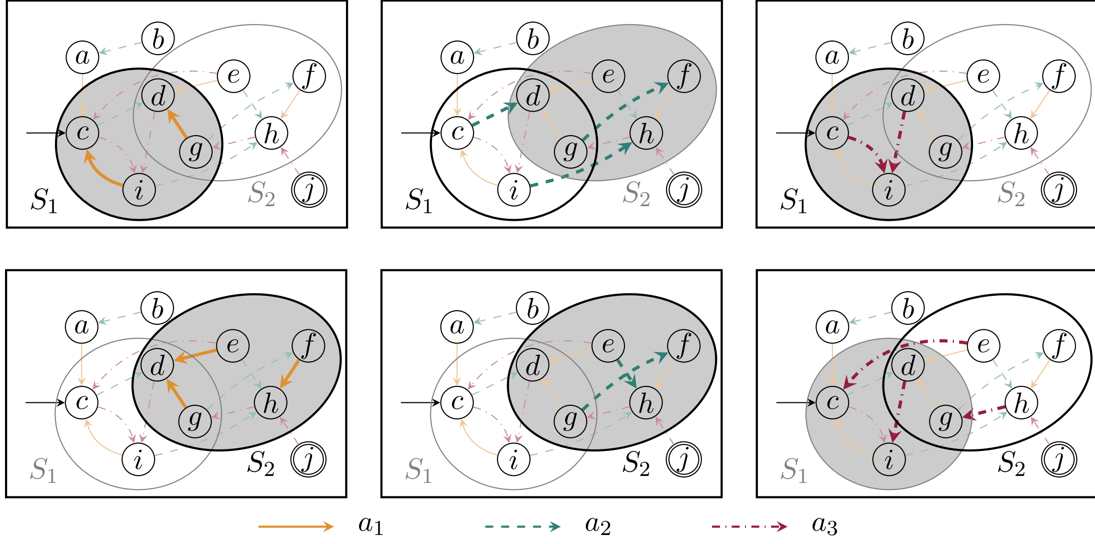



Figure 4.4: An example of a 1-disjunctive certificate in a search space with three actions. The top row highlights for $S_1$ and each action to which set the action application leads; the bottom row shows the same for $S_2$. The thick outlined area denotes the origin of the considered action, while the filled area shows the where the action leads.

**Proof:** Let $\mathcal{F}$ be an $r$-disjunctive certificate, and let $\mathcal{S} = \bigcup_{S \in \mathcal{F}} S$. We show that each of the three properties of Definition 4.5 (for $\mathcal{F}$) implies the corresponding property of Definition 4.3 (for $\mathcal{S}$). Properties 1 and 2 of the two definitions are equivalent by simple set arithmetic.

For property 3, consider any state $s \in \mathcal{S}$ and any action $a$ applicable in $s$. Because $\mathcal{S} = \bigcup_{S \in \mathcal{F}} S$, there exists a set $S \in \mathcal{F}$ with $s \in S$. From Definition 4.5, there must be a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ such that $S[a] \subseteq \bigcup_{S' \in \mathcal{F}'} S'$, so we get $S[a] \subseteq \bigcup_{S' \in \mathcal{F}'} S' \subseteq \bigcup_{S \in \mathcal{F}} S = \mathcal{S}$. From $s \in S$, we know that $s[a] \in S[a]$ and conclude that $s[a] \in \mathcal{S}$. As this is true for arbitrary states $s \in \mathcal{S}$ and applicable actions $a$, set $\mathcal{S}$ is inductive. $\square$

Figure 4.4 shows an example of a 1-disjunctive certificate $\mathcal{F} = \{S_1, S_2\}$ for a task with actions $a_1$, $a_2$ and $a_3$. Inclusion of the initial state $c$ can be seen from $c \in S_1$; exclusion of the only goal state $j$ from $j \notin S_1$ and $j \notin S_2$. For inductivity we first consider $S_1$: we have $S_1[a_1](= \{c, d\}) \subseteq S_1$, $S_1[a_2](= \{d, f, h\}) \subseteq S_2$ and $S_1[a_3](= \{i\}) \subseteq S_1$. For $S_2$ we have $S_2[a_1](= \{d, h\}) \subseteq S_2$, $S_2[a_2](= \{f, h\}) \subseteq S_2$ and $S_2[a_3](= \{c, g, i\}) \subseteq S_1$. As we can see we never needed to compute a union (or in other words we only needed unions of size 1), making the certificate 1-disjunctive. If for example $g[a_1] = e$ instead of $d$, $\mathcal{F}$ would still be a disjunctive certificate; but since $S_1[a_1]$ is now neither a subset of $S_1$ nor of $S_2$, it is no longer a 1-disjunctive certificate.

It is left to show that $r$-disjunctive certificates can indeed be verified efficiently. We define an ($r$-)disjunctive $\mathbf{R}$-certificate as an ($r$-)disjunctive certificate where all sets are represented as $\mathbf{R}$-formulas. In the following we show how we can verify that a set of

$\mathbf{R}$-formulas indeed represents an $r$-disjunctive certificate:

**Theorem 4.6.** *For fixed $r$, $r$-disjunctive $\mathbf{R}$-certificates $\Phi$ for task $\Pi$ can be verified in polynomial time in $\|\Phi\|$ and $\|\Pi\|$ if $\mathbf{R}$ efficiently supports **MO**, **CE**, $\wedge$**BC**, **CL**, **RN**$_\prec$ and either (a) $\vee$**BC** and **SE** or (b) **toCNF**.*

**Proof:** Let $\Phi$ be an $r$-disjunctive $\mathbf{R}$-certificate for task $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$. We must verify the three requirements of Definition 4.5.

For requirement 1 (initial state), we must verify that $I^\Pi \in \textit{states}(\varphi)$ for some $\varphi \in \Phi$. This can be done in polynomial time using **MO**, iterating over all $\varphi \in \Phi$.

For requirement 2 (non-inclusion of goal states), we must verify that $\textit{states}(\varphi) \cap S_G^\Pi = \emptyset$ for all $\varphi \in \Phi$. Each test can be performed using **CE**, as in the proof of Theorem 4.4.

For requirement 3 (disjunctive $r$-inductivity), we iterate for all $\varphi \in \Phi$, $a \in A^\Pi$ over all $\Phi' \subseteq \Phi$ with $|\Phi'| \leq r$ until we found a $\Phi'$ with $\textit{states}(\varphi)[a] \subseteq \bigcup_{\varphi' \in \Phi'} \textit{states}(\varphi')$. Since $r$ is fixed, there is only a polynomial number of such $\Phi'$.

Each test whether $\textit{states}(\varphi)[a] \subseteq \bigcup_{\varphi' \in \Phi'} \textit{states}(\varphi')$ can be performed following the same ideas as in the proof of Theorem 4.4, replacing $\varphi$ on the right side of the entailment with $\bigvee_{\varphi' \in \Phi} \varphi'$, resulting in the following formula:

$$(\varphi \wedge \bigwedge_{v_p \in \textit{pre}(a)} v_p)[(\textit{add}(a) \cup \textit{del}(a)) \to E'] \wedge \bigwedge_{v_a \in \textit{add}(a)} v_a \wedge \bigwedge_{v_d \in (\textit{del}(a) \setminus \textit{add}(a))} \neg v_d \models \bigvee_{\varphi' \in \Phi'} \varphi'$$

Expressing the left side of the entailment requires **CL**, **RN**$_\prec$ and $\wedge$**BC**. We can then build the disjunction on the right side directly with $\vee$**BC** and check entailment with **SE**. Alternatively we transform each $\varphi'$ into a CNF with **toCNF**. Building a single CNF from a disjunction of CNFs is exponential in the size of the disjunction, but for a fixed size (in our case $r$) polynomial. The entailment test can then be done by checking each clause of this CNF separately with **CE**. □

This means all considered formalisms are suitable. For BDDs we use $\vee$**BC** and **SE**. Horn and 2CNF formulas do not support $\vee$**BC**, but can use **toCNF** instead. For MODS we can either use $\vee$**BC** and **SE** if all formulas $\varphi'$ are over the same set of variables, otherwise we use **toCNF**.

## 4.4 Conjunctive Certificates

Analogously to disjunctive certificates, we can define conjunctive certificates as follows:

**Definition 4.6** (conjunctive certificate)**.** *A family $\mathcal{F} \subseteq 2^{S^\Pi}$ of state sets of task $\Pi$ is called a* conjunctive certificate *if $\bigcap_{S \in \mathcal{F}} S$ is an inductive certificate for $\Pi$.*

Similar to the disjunctive case, conjunctive certificates in themselves do not offer any advantage, and we thus define a restricted parameterized version with stronger properties. A notable difference however is that the property of non-inclusion of goal states also involves a subfamily of parameterized size.

**Definition 4.7** ($r$-conjunctive certificate)**.** *For $r \in \mathbb{N}_0$, a family $\mathcal{F} \subseteq 2^{S^\Pi}$ of state sets of task $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$ is called an $r$-*conjunctive certificate *if:*

1. *$I^\Pi \in S$ for all $S \in \mathcal{F}$,*
2. *there is a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ with $|\mathcal{F}'| \leq r$ such that $(\bigcap_{S \in \mathcal{F}'} S) \cap S_G^\Pi = \emptyset$, and*
3. *for all $S \in \mathcal{F}$ and all $a \in A^\Pi$, there is a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ with $|\mathcal{F}'| \leq r$ such that $(\bigcap_{S' \in \mathcal{F}'} S')[a] \subseteq S$.*

We call the third property of this definition *conjunctive $r$-inductivity*. As in the disjunctive case, we can show that an $r$-conjunctive certificate is a proof of unsolvability:

**Theorem 4.7.** *All $r$-conjunctive certificates are conjunctive certificates.*

**Proof:** Let $\mathcal{F}$ be an $r$-conjunctive certificate, and let $\mathcal{S} = \bigcap_{S \in \mathcal{F}} S$. We show that each of the three properties of Definition 4.7 (for $\mathcal{F}$) implies the corresponding property of Definition 4.3 (for $\mathcal{S}$). Property 1 is equivalent, and it is easy to see that property 2 of Definition 4.7 implies property 2 of Definition 4.3.

For property 3, consider any state $s \in \mathcal{S}$ and any action $a$ applicable in $s$. We will show that $s[a]$ is in *all* sets $S \in \mathcal{F}$ and hence also in their intersection $\mathcal{S}$. For this purpose, consider an arbitrary $S^* \in \mathcal{F}$. From Definition 4.7, there is a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ with $(\bigcap_{S \in \mathcal{F}'} S)[a] \subseteq S^*$. Since $\mathcal{S} = \bigcap_{S \in \mathcal{F}} S$, $s$ is contained in all $S \in \mathcal{F}$, and hence also in $\bigcap_{S \in \mathcal{F}'} S$. Therefore $s[a] \in (\bigcap_{S \in \mathcal{F}'} S)[a] \subseteq S^*$. □

Figure 4.5 shows an example of a 2-conjunctive certificate $\mathcal{F} = \{S_1, S_2, S_3\}$. The inclusion of the initial state $f$ can be seen from $f \in S_1$, $f \in S_2$ and $f \in S_3$. Exclusion of the goal states $i$ and $k$ can be seen from $S_1 \cap S_2 \cap \{i, k\} = \emptyset$. Note that we need an intersection of $S_1$ and $S_2$ here, since $i \in S_1$, $i \in S_3$, $k \in S_2$ and $k \in S_3$. For inductivity we can see for each set $S \in \mathcal{F}$ and each action $a \in \{a_1, a_2\}$ that applying $a$ to some intersection of at most two sets from $\mathcal{F}$ leads only to $S$. For $S_1$ we actually do not even need intersections since we can use $S_1[a_1] \subseteq S_1$ and $S_3[a_2] \subseteq S_1$ (depicted on the left side of Figure 4.5). For $S_2$ we see in the middle of Figure 4.5 that $(S_2 \cap S_3)[a_1] \subseteq S_2$ and $S_2[a_2] \subseteq S_2$. Note that for $a_1$ either only $S_2$ or only $S_3$ would not be sufficient because of transitions $h[a_1] = n$ and $m[a_1] = n$, and $S_1$ is not sufficient because of $i[a_1] = e$. Finally for $S_3$ we see on the right side of Figure 4.5 that $(S_1 \cap S_2)[a_1] \subseteq S_3$ and $(S_1 \cap S_3)[a_2] \subseteq S_3$. Since we only needed intersections of size 1 or 2, $\mathcal{F}$ is 2-conjunctive.

Analogously to our earlier terminology for disjunctive certificates, an ($r$-)conjunctive $\mathbf{R}$-certificate is an ($r$-)conjunctive certificate where the component sets are represented as $\mathbf{R}$-formulas. We can efficiently verify $r$-conjunctive $\mathbf{R}$-certificates, if they support the following operations efficiently:

**Theorem 4.8.** *For fixed $r$, $r$-conjunctive $\mathbf{R}$-certificates $\Phi$ for task $\Pi$ can be verified in polynomial time in $\|\Phi\|$ and $\|\Pi\|$ if $\mathbf{R}$ efficiently supports **MO**, **CE**, **SE**, $\wedge$**BC**, **CL** and **RN**$_\prec$.*

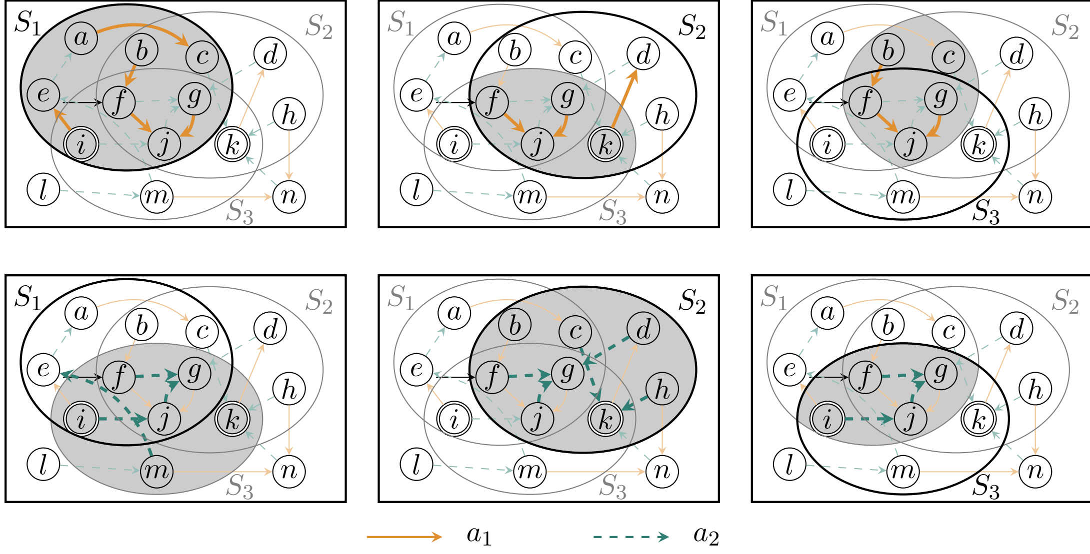



Figure 4.5: An example of a 2-conjunctive certificate in a search space with two actions. The filled area denotes the origin of the considered action, while the thick outlined area shows the where the action leads.

**Proof:** Let $\Phi$ be an $r$-conjunctive **R**-certificate for task $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi\rangle$. We must verify the three requirements of Definition 4.7.

For requirement 1 (initial state), we must verify that $s \in \mathit{states}(\varphi)$ for all $\varphi \in \Phi$. This can be done in polynomial time using **MO**, iterating over all $\varphi \in \Phi$.

For requirement 2 (non-inclusion of goal states), we must verify that there exists some $\Phi' \subseteq \Phi$ with $|\Phi'| \leq r$ such that $(\bigcap_{\varphi\in\Phi'} \mathit{states}(\varphi)) \cap S_G^\Pi = \emptyset$. We iterate over the $O(|\Phi|^r)$ candidates subsets until we found a suitable one. For each subset, we use a similar idea as in the proof of Theorem 4.4 and test whether $\bigwedge_{\varphi\in\Phi'} \varphi \models \bigvee_{v\in G^\Pi} \neg v$. While building $\bigwedge_{\varphi\in\Phi'} \varphi$ can be exponential in $r$, for fixed $r$ the runtime is polynomial if $\wedge$**BC** is efficiently supported. The entailment can then be checked with **CE**.

For requirement 3 (conjunctive $r$-inductivity), we must verify for all $\varphi \in \Phi$ and all $a \in A^\Pi$ that there exists some $\Phi' \subseteq \Phi$ with $|\Phi'| \leq r$ s.t. $(\bigcap_{\varphi'\in\Phi'} \mathit{states}(\varphi'))[a] \subseteq \mathit{states}(\varphi)$. We test this separately for each $\varphi$, $a$ and $\Phi'$, amounting to a polynomial number of tests. Applying the same ideas as Theorem 4.4 and 4.3 to avoid the forget operator and use **RN**$_\prec$ instead of **RN**, each test needs to check the following entailment with $\wedge$**BC**, **CL**, **RN**$_\prec$ and **SE**:

$$(\bigwedge_{\varphi'\in\Phi'} \varphi' \wedge \bigwedge_{v_p\in \mathit{pre}(a)} v_p)[(\mathit{add}(a) \cup \mathit{del}(a)) \to E'] \wedge \bigwedge_{v_a\in \mathit{add}(a)} v_a \wedge \bigwedge_{v_d\in(\mathit{del}(a)\setminus \mathit{add}(a))} \neg v_d \models \varphi$$

□

# 5 Unsolvability Proof System

A disadvantage of the inductive certificates presented in Chapter 4 is that they are *monolithic*, meaning that the planning system needs to find one set (possibly represented in a composite form) that is inductive. State of the art planning systems however commonly employ a variety of techniques which all reason in different ways. Even if we were able to build an inductive certificate for each of these techniques when they are used on their own, we are likely not able to build one if the techniques are used together. A simple reason is that different techniques might require different formalisms to represent their sets. The more fundamental problem is that when combined, one technique only contributes part of the reason why the task is unsolvable, and building an inductive set with these parts is not straightforward.

The approach presented in this chapter enables a much higher degree of composability by introducing a specialized proof system based on natural deduction. The advantage of this proof system is that it serves as a collection of knowledge we gather about the task, and once a piece of knowledge has been proven it can be used without concern about the details of the proof. This enables us to first prove the correctness of the reasoning of each applied technique by itself and then combine the learned knowledge in into the final argument why a task is unsolvable.

In Section 5.1 we will give a more formal background on natural deduction. Section 5.2 then defines the proof system. Afterwards, Section 5.3 analyzes under which circumstances proofs in this system can be efficiently verified. In Section 5.4 we compare the proof system to our previous approach of inductive certificates.

## 5.1 Natural Deduction

Natural deduction was formalized by Gentzen (1935) as an alternative to deductive reasoning. His main motivation was to provide a formal way of reasoning that more closely resembles the way of human reasoning. A proof system based on natural deduction is defined by its inference rules. An inference rule concludes that if certain assumptions are true, its conclusion is true as well. In propositional logic for example, given the assumption that $A \wedge B$ is true we can conclude that $A$ is true. Inference rules are universal in the sense that they do not talk about concrete objects, e.g. $A$ and $B$ in the above example are not concrete formulas but placeholders for an arbitrary formula. The correctness of a new inference rule must be proven either *outside* of the proof system (for example with a handcrafted proof) or by using already existing rules, but once proven correct it can be

used universally within the proof system for any instantiation of its placeholders.

In natural deduction we reason about *judgments*. A judgment is something which can be known, e.g. "It rains.". An *inference rule* takes a certain amount of judgments as premises and concludes a new judgment from it. For example we could say "If it rains, it is cloudy.", and could use this inference rule when it is raining to deduce that it must be cloudy.

An inference rule is defined by a name, a number of *premises* $A_1, \ldots, A_n$ and a *conclusion* $B$. If the inference rule has no premises, it is called an *axiom*. We will represent inference rules in the following way:

$$\frac{A_1 \qquad \ldots \qquad A_n}{B}\ \text{name}$$

When using an inference rule during a proof, the premises are either proven through other rules (e.g. axioms) or are (temporarily) assumed. Rules can also discharge previously made assumptions. To illustrate this consider the implication operator in propositional logic: If one can prove that, under the assumption that $\phi$ holds, a certain proposition $\psi$ holds (possibly with several steps), then $\phi \to \psi$ must hold, even if $\phi$ does not. We denote the discharging of assumptions by adding square brackets to the discharged assumption and add a superscript to it and the rule name:

$$\begin{array}{c} [\Phi]^i \\ \vdots \\ \dfrac{\Psi}{\Phi \to \Psi} \to I^i \end{array}$$

Since proof systems often talk about different types of objects, placeholder variables such as $\Phi$ and $\Psi$ are often identified with a *type*. In the case of propositional logic, we could for example define the types "variable" ($X := x_i$) and "proposition" ($P := X|\neg P|P \wedge P|P \vee P$). In this rule, $\Phi$ and $\Psi$ are propositions. To denote that $\Phi$ is a proposition, we write "$\Phi : P$".

Below is an example of the general structure of a proof. The proof uses two rules from propositional logic: the previously defined rule $\to I$, as well as a rule $\wedge L$ with premise $\Phi \wedge \Psi$ and conclusion $\Phi$.

$$\dfrac{\dfrac{[\varphi \wedge \psi]^1}{\varphi}\ \wedge L}{\varphi \wedge \psi \to \varphi} \to I^1$$

Since there are no open assumptions in the proof, we have shown that $\varphi \wedge \psi \to \varphi$ is always true.

## 5.2 Proof System for Unsolvability of Planning Tasks

A witness in this approach is a proof for unsolvability of a planning task within our proof system. As we stated in Chapter 1, we aim for witnesses that are sound and complete. This means the proof system must be sound itself, i.e. if something is proven within the proof system it must be true. It does however not need to be complete: A complete proof system is defined as a system where we can prove everything that is true. We only want completeness with respect to proving tasks unsolvable, i.e. if a task is unsolvable we can prove it within the proof system. We thus instead say that our proof system is sound, and complete with respect to the judgment "task unsolvable.".

Our motivation for introducing a proof system was to allow for more composite forms of reasoning. The reasoning planning techniques perform usually revolves around the idea to show that certain areas of the search space cannot be part of a solution, either because we cannot reach a goal from it or because we cannot reach it from the initial state. To formalize this concept we introduce the notion of dead states and dead state sets.

**Definition 5.1** (dead state and dead state set)**.** *A state $s$ is dead if no plan traverses $s$, or more precisely there is no plan $\pi = \langle a_1, \ldots a_n \rangle$ and $1 \leq i \leq n$ with $s = I[a_1] \ldots [a_i]$. A set of states is dead if all its elements are dead.*

In general, deciding whether a state or a state set is dead is **PSPACE**-complete, since the planning problem (which is **PSPACE**-complete) can be reformulated to the question "Is $I^\Pi$ dead?". There are however many cases where deciding deadness is efficiently possible. For example, any non-goal state with no successors is dead, as well as any state other than the initial state with no predecessors.

We will not consider single states, but instead focus on sets of states $S$. We call these sets *state set expressions* and define them in the following ways:

- explicitly in a formalism $\mathbf{R}$,
- abstractly as a complement, union, intersection, progression or regression of state set expressions,
- or as a constant $\{I^\Pi\}$, $S_G^\Pi$ or $\emptyset$.

We call a state set defined explicitly or as a constant a *state set variable*, and a state set variable or its negation a *state set literal*.

Additionally we consider action set expressions $A$ which are defined either explicitly as an enumeration of actions, abstractly as the union of action set expressions, or as the constant $A^\Pi$.

### 5.2.1 Inference Rules

The inference rules in the proof system focus on proving state sets dead. We first define some basic rules whose correctness can be seen immediately:

- $\emptyset$ is dead.
- If $S$ and $S'$ are dead, then $S \cup S'$ is dead.
- If $S$ is dead and $S' \subseteq S$, then $S'$ is dead.

Next we want to define two special rules that finish a proof by stating that the task is unsolvable:

**Theorem 5.1.** *If the initial state is dead or all goal states are dead, then the task is unsolvable.*

**Proof:** If the initial state is dead, no action sequence can be a plan because every plan must traverse the initial state. If all goal states are dead, no action sequence can be a plan because every plan must traverse a goal state. □

The next two theorems form the centerpiece of our proof system, as they introduce the only rules that can deduce new parts of the state space as dead.

**Theorem 5.2.** *Let $S$ be a set of states and $S'$ a dead set of states such that all successors of states $s \in S$ are in $S$ or $S'$, i.e. , $S[A^\Pi] \subseteq S \cup S'$.*

*(i) If $S \cap S_G^\Pi$ is dead, then $S$ is dead.*

*(ii) If $I^\Pi \in S$, then $\overline{S}$ is dead.*

**Proof:** For any plan $\pi = \langle a_1, \ldots, a_n \rangle$ traversing a state from $S$ we can show that the plan will never leave $S$ again. More formally, if $I^\Pi[\langle a_1, \ldots, a_i \rangle] \in S$ for some $i \leq n$, then $I^\Pi[\langle a_1, \ldots, a_j \rangle] \in S$ for all $i \leq j \leq n$.

For $i = n$ we have nothing to show. Otherwise, consider $s = I^\Pi[\langle a_1, \ldots, a_{i+1} \rangle]$. From $S[A^\Pi] \subseteq S \cup S'$ we know that $s \in S \cup S'$. But since $S'$ is dead and $s$ is part of a plan and thus cannot be dead, we conclude that $s \in S$. Applying this argument iteratively proves the claim.

(i) By contradiction: assume that $S \cap S_G^\Pi$ is dead and $S$ is not dead. Then a plan $\pi$ traversing a state in $S$ exists. We have shown that $I^\Pi[\pi] \in S$. Because $\pi$ is a plan, we also get that $I^\Pi[\pi] \in S_G^\Pi$ and that $I^\Pi[\pi]$ is not dead. This contradicts the fact that $S \cap S_G^\Pi$ is dead.

(ii) From $I^\Pi \in S$ we know that no plan leaves $S$, and hence $s \notin S$ cannot be traversed by a plan. Hence $\overline{S}$ is dead. □

Figure 5.1 illustrates these rules. On the left side, we see that all goal states contained in $S$ are dead. This means that in order to reach a goal from $S$ we need to leave $S$. But this leads us to $S'$, which cannot be part of a plan. Thus any state in $S$ cannot be part of a plan either. On the right side, the initial state is in $S$, meaning all plans must start in $S$. Since we can only leave $S$ though dead states, we can conclude that the entire plan must lie in $S$ and thus all states outside of $S$ are dead.

For regression, we can formulate similar rules:

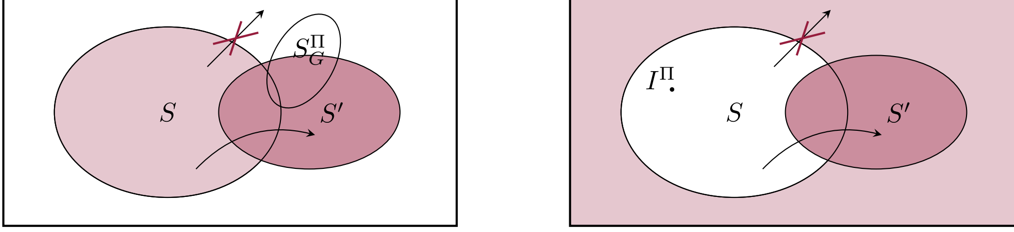



Figure 5.1: Illustration of the inference rules related to progression. $S'$ is already proven to be dead, and successors of $S$ are either in $S$ or $S'$.

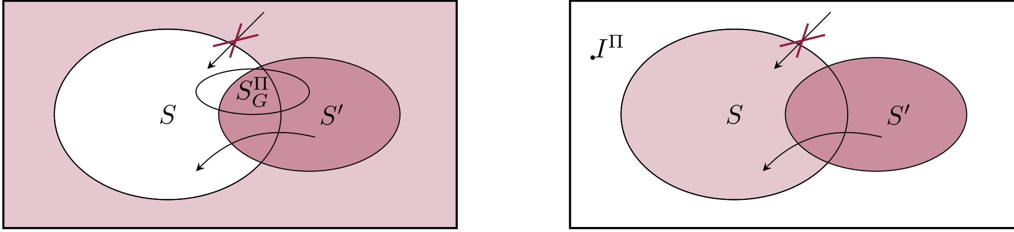



Figure 5.2: Illustration of the inference rules related to regression. $S'$ is already proven to be dead, and predecessors of $S$ are either in $S$ or $S'$.

**Theorem 5.3.** *Let $S$ be a set of states and $S'$ a dead set of states such that all predecessors of states $s \in S$ are in $S$ or $S'$, i.e. , $[A^\Pi]S \subseteq S \cup S'$.*

*(i) If $\overline{S} \cap S_G^\Pi$ is dead, then $\overline{S}$ is dead.*

*(ii) If $I^\Pi \notin S$, then $S$ is dead.*

**Proof:** Similarly to progression, we can argue that if a plan $\pi = \langle a_1, \dots, a_n \rangle$ traverses a state from $S$, it cannot have traversed a state from $\overline{S}$ earlier, or more formally: if $I^\Pi[\langle a_1, \dots, a_i \rangle] \in S$ for some $i \leq n$, then for all $j \leq i$ we get $I^\Pi[\langle a_1, \dots, a_j \rangle] \in S$ and in particular $I^\Pi[\langle\rangle] = I^\Pi \in S$.

(i) If $\overline{S} \cap S_G^\Pi$ is dead, then all plans must end in a goal state in $S$. With the above argument the plans traverse only states from $S$, so $\overline{S}$ is dead.

(ii) If a plan traverses a state from $S$, then $I^\Pi$ is in $S$. As $I^\Pi \notin S$, there cannot be such a plan, and hence $S$ is dead. □

Figure 5.2 illustrates the regression rules. On the left, we see that any goal state that can be part of a plan (i.e. that is not dead) is contained in $S$, thus all plans must end in $S$. Since we can only enter $S$ over $S'$, but $S'$ cannot be part of a plan, the entire plan must be contained in $S$. On the right, all plans must start outside of $S$ since $I^\Pi \notin S$. In order for a plan to get to $S$ it must go through $S'$, but no plan can do this; thus $S$ cannot be part of a plan.

We also include a rule with relates progression and regression and is a generalization of Theorem 4.2:

**Theorem 5.4.** $S[A] \subseteq S'$ *iff* $[A]\overline{S'} \subseteq \overline{S}$.

**Proof:** If $S[A] \subseteq S'$, then all transitions originating from a state in $S$ with $a \in A$ lead to a state in $S'$. This means that for all states not in $S'$ (i.e. , $s \in \overline{S'}$) their predecessors with $a$ cannot lie in $S$ and hence $[A]\overline{S'} \subseteq \overline{S}$. If $[A]\overline{S'} \subseteq \overline{S}$, then all transitions leading to a state in $\overline{S'}$ with $a \in A$ originate from a state in $\overline{S}$. This means that for all states in $S$, their successors with $a$ must lie in $S'$ and hence $S[A^{\Pi}] \subseteq S'$. □

The rules for progression and regression can also be derived from each other with the help of Theorem 5.4. To illustrate this, we show how rule (i) from Theorem 5.3 can be proven by applying rule (i) from Theorem 5.2, Theorem 5.4 and basic set rules:

We are given two sets $S_1$ and $S_2$, knowing that (1) $[A^{\Pi}]S_1 \subseteq S_1 \cup S_2$, (2) $S_2$ is dead and (3) $\overline{S_1} \cap S_G^{\Pi}$ is dead. From this knowledge we want to show that $\overline{S_1}$ is dead. We first define $X_1 = \overline{S_1} \cap \overline{S_2}$, and $X_2 = \overline{S_1} \cap S_2$. With (1) and Theorem 5.4 we know that $\overline{S_1 \cup S_2}[A^{\Pi}] \subset \overline{S_1}$. Since $\overline{S_1 \cup S_2} = X_1$ and $\overline{S_1} = (\overline{S_1} \cap \overline{S_2}) \cup (\overline{S_1} \cap S_2) = X_1 \cup X_2$ we can reformulate this to (4) $X_1[A^{\Pi}] = X_1 \cup X_2$. Next, from $S_2$ being dead and $X_2 \subseteq S_2$ we know that (5) $X_2$ is dead as well (since a subset of a dead set is dead). From (3) we know that $\overline{S_1} \cap \overline{S_2} \cap S_G^{\Pi}$ must be dead for the same reason, and can reformulate this to (6) $X_1 \cap S_G^{\Pi}$ is dead. Now we can apply rule (i) from Theorem 5.2 with (4),(5),(6) with the result that $X_1 = \overline{S_1} \cap \overline{S_2}$ is dead. Since we know already that $S_2$ is dead and that the union of two dead state sets is dead as well, we can conclude that $(\overline{S_1} \cap \overline{S_2}) \cup S_2 = \overline{S_1} \cup S_2$ is dead, and thus $\overline{S_1}$ must also be dead.

We also include a number of rules from set theory in order to be able to argue effectively with sets. We only include rules that we need later on for specific use cases in order to keep the proof system minimal; but if future work requires more rules they can always be integrated. We will not discuss these rules here since they are already well established, but they are listed in the next section.

Finally, we introduce a number of rules related to progression, as we need them later to show the connection between our proof system and inductive certificates. Similar rules could be introduced for regression as well, but as with the set theory rules above we only include rules that we currently need.

The first two rules follow directly from the monotonicity of progression:

- If $S[A] \subseteq S'$ and $A' \subseteq A$, then $S[A'] \subseteq S'$
- If $S[A] \subseteq S'$ and $S'' \subseteq S$, then $S''[A] \subseteq S'$

The other two rules follow from the definition of progression:

- If $S[A] \subseteq S'$ and $S[A'] \subseteq S'$, then $S[A \cup A'] \subseteq S'$.
- If $S[A] \subseteq S'$ and $S''[A] \subseteq S'$, then $(S \cup S'')[A] \subseteq S'$.

For the first case, consider any state $s \in S$ and $a \in (A \cup A')$. If $a \in A$, then $s[a]$ must be in $S'$ from $S[A] \subseteq S'$, and if $a \in A'$ the same can be concluded form $S[A'] \subseteq S'$. For the second case, consider any state $s \in (S \cup S')$ and $a \in A$. If $s \in S$, then from $S[A] \in S'$ follows $s[a] \in S'$, and the case $s \in S'$ is analogous.

### 5.2.2 Basic Statements

The inference rules discussed above all operate on a purely syntactical level, meaning they do not interpret the actual semantics of the involved sets, and depend on judgments we already know about them. As a result, we lack a starting point for gathering judgments. For example, consider a planning task with state $s$ with no successor. It is easy to see that judgment $\{s\}[A^{\Pi}] \subseteq \{s\}$ is true, but we cannot prove it within the proof system. To overcome this problem, we define an additional source of judgments that interprets the semantics of the sets and thus must be proven separately in each proof. A judgment that must be proven this way is called a *basic statement*.

We only consider basic statements in the form of $S \subseteq S'$. Verifying this statement can become a difficult task, since the definition of a state set can be arbitrarily complex and use different formalisms. Instead we limit the statement to practically useful cases and as general as possible, but can be verified in polynomial time if the involved formalisms support certain operations efficiently. In what follows, $X_{\mathbf{R}}$ is a state set variable represented by formalism $\mathbf{R}$, $L_{\mathbf{R}}$ a set literal where the underlying state set variable is represented by formalism $\mathbf{R}$, and $r$ is a constant bounding the size of intersections and unions. We consider the following statements:

**1.** $\bigcap_{L_{\mathbf{R}} \in \mathcal{L}} L_{\mathbf{R}} \subseteq \bigcup_{L'_{\mathbf{R}} \in \mathcal{L}'} L'_{\mathbf{R}}$ **with** $|\mathcal{L}| + |\mathcal{L}'| \leq r$ The first basic statement restricts $S$ to be an intersection of state set literals and $S'$ a union of state set literals, where all involved state set variables are represented with the same formalism $\mathbf{R}$. This includes the special case where the intersection and/or union contains only one element, i.e. $S$ and/or $S'$ are a set literal.

**2.** $(\bigcap_{X_{\mathbf{R}} \in \mathcal{X}} X_{\mathbf{R}})[A] \cap \bigcap_{L_{\mathbf{R}} \in \mathcal{L}} L_{\mathbf{R}} \subseteq \bigcup_{L'_{\mathbf{R}} \in \mathcal{L}'} L'_{\mathbf{R}}$ **with** $|\mathcal{X}| + |\mathcal{L}| + |\mathcal{L}'| \leq r$ The second basic statement adds progression. $S'$ is restricted the same way as in the first basic statement, but the intersection in $S$ can now also contain the progression of an intersection of state variables. Note that the progression cannot contain state literals.

**3.** $[A](\bigcap_{X_{\mathbf{R}} \in \mathcal{X}} X_{\mathbf{R}}) \cap \bigcap_{L_{\mathbf{R}} \in \mathcal{L}} L_{\mathbf{R}} \subseteq \bigcup_{L'_{\mathbf{R}} \in \mathcal{L}'} L'_{\mathbf{R}}$ **with** $|\mathcal{X}| + |\mathcal{L}| + |\mathcal{L}'| \leq r$ The third basic statement adds regression. It is identical to the second basic statement except that progression is replaced with regression.

**4.** $L_{\mathbf{R}} \subseteq L'_{\mathbf{R}'}$ The fourth basic statement restricts $S$ and $S'$ to only a set literal, but instead allows to use different formalisms for $S$ and $S'$. This is important for being able to combine knowledge gained from techniques requiring different formalisms.

**5.** $A \subseteq A'$ The final basic statement considers actions instead of sets. Action set expressions $A$ and $A'$ are not restricted, i.e. they can be defined as either an explicit enumeration, a union or the constant set $A^{\Pi}$.

### 5.2.3 Overview

In what follows we give a full overview of the components of the proof system. We first define the type of objects used, then list the inference rules used within the proof system and afterwards the basic statements which have to be verified based on semantics. To emphasize that the proof system interprets judgments on a purely syntactical level, we write $S \sqsubseteq S$ instead of $S \subseteq S$ when denoting a judgment in the proof system.

**Types** We define the object types through a simple grammar:

| | |
|---|---|
| state set variables | $X := \{I^\Pi\} \mid S_G^\Pi \mid \emptyset \mid X_\mathbf{R}$ |
| state set literals | $L := X \mid \overline{X}$ |
| state set expressions | $S := L \mid (S \cup S) \mid (S \cap S) \mid S[A] \mid [A]S$ |
| action set expressions | $A := A^\Pi \mid a \mid (A \cup A)$ |
| set expressions | $E := S \mid A$ |

Set expressions are defined separately to enable us to define basic set theory rules that can be applied to both state set expressions and action set expressions. In what follows we denote an object of e.g. type $E$ by simply $E$,$E'$ or $E''$ instead of $Z : E$ and also do not mention the type of constant expressions since they are defined above.

**Rules** The following rules show that state sets are dead:

**E**mpty set **D**ead $$\frac{}{\emptyset \text{ dead}}\ \text{ED}$$

**U**nion **D**ead $$\frac{S \text{ dead} \qquad S' \text{ dead}}{S \cup S' \text{ dead}}\ \text{UD}$$

**S**ubset **D**ead $$\frac{S' \text{ dead} \qquad S \sqsubseteq S'}{S \text{ dead}}\ \text{SD}$$

**P**rogression **G**oal $$\frac{S[A^\Pi] \sqsubseteq S \cup S' \qquad S' \text{ dead} \qquad S \cap S_G^\Pi \text{ dead}}{S \text{ dead}}\ \text{PG}$$

**P**rogression **I**nitial $$\frac{S[A^\Pi] \sqsubseteq S \cup S' \qquad S' \text{ dead} \qquad \{I^\Pi\} \sqsubseteq S}{\overline{S} \text{ dead}}\ \text{PI}$$

**R**egression **G**oal $$\frac{[A^\Pi]S \sqsubseteq S \cup S' \qquad S' \text{ dead} \qquad \overline{S} \cap S_G^\Pi \text{ dead}}{\overline{S} \text{ dead}}\ \text{RG}$$

**R**egression **I**nitial $$\frac{[A^\Pi]S \sqsubseteq S \cup S' \qquad S' \text{ dead} \qquad \{I^\Pi\} \sqsubseteq \overline{S}}{S \text{ dead}}\ \text{RI}$$

These rules show that the task is unsolvable:

**C**onclusion **I**nitial $$\frac{\{I^\Pi\} \text{ dead}}{\text{unsolvable}}\ \text{CI}$$

**C**onclusion **G**oal $$\frac{S_G^\Pi \text{ dead}}{\text{unsolvable}}\ \text{CG}$$

These rules from basic set theory can be used for both state and action set expressions:

**U**nion **R**ight $\dfrac{}{E \sqsubseteq (E \cup E')}$ UR

**U**nion **L**eft $\dfrac{}{E \sqsubseteq (E' \cup E)}$ UL

**I**ntersection **R**ight $\dfrac{}{(E \cap E') \sqsubseteq E}$ IR

**I**ntersection **L**eft $\dfrac{}{(E' \cap E) \sqsubseteq E}$ IL

**DI**stributivity $\dfrac{}{((E \cup E') \cap E'') \sqsubseteq ((E \cap E'') \cup (E' \cap E''))}$ DI

**S**ubset **U**nion $\dfrac{E \sqsubseteq E'' \qquad E' \sqsubseteq E''}{(E \cup E') \sqsubseteq E''}$ SU

**S**ubset **I**ntersection $\dfrac{E \sqsubseteq E' \qquad E \sqsubseteq E''}{E \sqsubseteq (E' \cap E'')}$ SI

**S**ubset **T**ransitivity $\dfrac{E \sqsubseteq E' \qquad E' \sqsubseteq E''}{E \sqsubseteq E''}$ ST

The final rules focus on progression and its relation to regression:

**A**ction **T**ransitivity $\dfrac{S[A] \sqsubseteq S' \qquad A' \sqsubseteq A}{S[A'] \sqsubseteq S'}$ AT

**A**ction **U**nion $\dfrac{S[A] \sqsubseteq S' \qquad S[A'] \sqsubseteq S'}{S[A \cup A'] \sqsubseteq S'}$ AU

**P**rogression **T**ransitivity $\dfrac{S[A] \sqsubseteq S'' \qquad S' \sqsubseteq S}{S'[A] \sqsubseteq S''}$ PT

**P**rogression **U**nion $\dfrac{S[A] \sqsubseteq S'' \qquad S'[A] \sqsubseteq S''}{(S \cup S')[A] \sqsubseteq S''}$ PU

**P**rogression to **R**egression $\dfrac{S[A] \sqsubseteq S'}{[A]\overline{S'} \sqsubseteq \overline{S}}$ PR

**R**egression to **P**rogression $\dfrac{[A]\overline{S'} \sqsubseteq \overline{S}}{S[A] \sqsubseteq S'}$ RP

**Basic Statements**

**B1** $\bigcap_{L_{\mathbf{R}} \in \mathcal{L}} L_{\mathbf{R}} \subseteq \bigcup_{L'_{\mathbf{R}} \in \mathcal{L}'} L'_{\mathbf{R}}$ with $|\mathcal{L}| + |\mathcal{L}'| \leq r$

**B2** $(\bigcap_{X_{\mathbf{R}} \in \mathcal{X}} X_{\mathbf{R}})[A] \cap \bigcap_{L_{\mathbf{R}} \in \mathcal{L}} L_{\mathbf{R}} \subseteq \bigcup_{L'_{\mathbf{R}} \in \mathcal{L}'} L'_{\mathbf{R}}$ with $|\mathcal{X}| + |\mathcal{L}| + |\mathcal{L}'| \leq r$

**B3** $[A](\bigcap_{X_{\mathbf{R}} \in \mathcal{X}} X_{\mathbf{R}}) \cap \bigcap_{L_{\mathbf{R}} \in \mathcal{L}} L_{\mathbf{R}} \subseteq \bigcup_{L'_{\mathbf{R}} \in \mathcal{L}'} L'_{\mathbf{R}}$ with $|\mathcal{X}| + |\mathcal{L}| + |\mathcal{L}'| \leq r$

**B4** $L_{\mathbf{R}} \subseteq L'_{\mathbf{R}'}$

**B5** $A \subseteq A'$

## 5.3 Efficient Verification

Any application of an inference rule is inherently efficient, since we only need to check that the premises are known and that the rule is applied properly on a syntactical level. For example, a step in the proof might claim that $\overline{S_2} \cup (S_4 \cap \overline{S_6})$ is dead based on applying rule UD to premises "$\overline{S_2}$ is dead" and "$(S_4 \cap \overline{S_6})$ is dead". The verifier only needs to check if the premises are known judgments, if rule UD is applicable to the premises and if the conclusion of applying the rule matches the conclusion the proof claims. This is efficient since we require syntactical equality. For example, if the second premise instead were "$(\overline{S_6} \cap S_4)$ is dead", the verifier would reject the proof even if semantically the claim is correct, because it is not equal on the syntactical level. Similarly, if the proof claims conclusion "$(S_4 \cap \overline{S_6}) \cup \overline{S_2}$ is dead" when having "$\overline{S_2}$ is dead" as first and "$(S_4 \cap \overline{S_6})$ is dead" as second argument, the verifier will reject the proof.

Basic Statements on the other hand require to analyze the semantics of the expressions used in the statements. Basic statement **B5** $A \subseteq A'$ is trivial to verify, since we can retrieve all elements of $A$ and $A'$ in time linear in their representation size or the representation of the planning task: If $A$ ($A'$) is defined explicitly, all elements are represented explicitly; if $A$ ($A'$) is the constant $A^\Pi$, we retrieve the actions from planning tasks; and if $A$ ($A'$) is a union of $A_1$ and $A_2$, we can first retrieve all elements of $A_1$ and $A_2$ and then merge the two explicit sets.

For state set expressions, the semantics of a state set variable $X$ is given by an $\mathbf{R}$-formula $\varphi_\mathbf{R}$ over variables $V$, such that the models of $\varphi_\mathbf{R}$ represent the states in $X$. The semantics of a constant variable can be represented by any formalism $\mathbf{R}$ that supports **CL**. Composite set expressions are represented based on their underlying state set variables and using the connective $\vee$ for $\cup$, $\wedge$ for $\cap$, $\neg$ for complement and the formulas in Definition 3.15 and 3.16 for progression and regression. To formalize this, we define the interpretation function $I$ which maps set expressions to logical formulas in the following way:

$I(X) := \varphi_\mathbf{R}$ where $X$ is represented as $\varphi_\mathbf{R}$

$I(\{I^\Pi\}) := \bigwedge_{v \in I^\Pi} v \wedge \bigwedge_{v' \notin I^\Pi} \neg v'$

$I(S_G^\Pi) := \bigwedge_{g \in G^\Pi} g$

$I(\emptyset) := v \wedge \neg v$ for some $v \in V^\Pi$

$I(\overline{S}) := \neg I(S)$

$I(S \cup S) := I(S) \vee I(S')$

$I(S \cap S) := I(S) \wedge I(S')$

$I(S[A]) := \bigvee_{a \in A} \Big(\exists(\mathit{add}(a) \cup \mathit{del}(a)).(I(S) \wedge \bigwedge_{v_p \in \mathit{pre}(a)} v_p)\Big) \wedge \bigwedge_{v_a \in \mathit{add}(a)} v_a$
$\wedge \bigwedge_{v_d \in (\mathit{del}(a) \setminus \mathit{add}(a))} \neg v_d$

$I([A]S) := \bigvee_{a \in A} \Big( \exists (add(a) \cup del(a)).(I(S) \land \bigwedge_{v_a \in add(a)} v_a \land \bigwedge_{v_d \in (del(a) \setminus add(a))} \neg v_d) \Big)$
$\land \bigwedge_{v_p \in pre(a)} v_p$

A subset relation $S \subseteq S'$ now holds iff $I(S) \models I(S')$ does.

Note that the interpretation does not consider the formalism used in concrete formulas. We can express a formula only if the formulas representing the underlying state set variables are all in the same formalism.

### 5.3.1 Single Representation

We will first analyze basic statements where all involved state set variables are represented by the same formalism $\mathbf{R}$. We do this in an incremental way, starting with the case where both the left and right side consist of only one state set variable. As we add unions and intersections, we will assume that the total amount of set variables occurring in a statement is bounded by $r$. Since all formulas are encoded in $\mathbf{R}$ we will write $\varphi$ instead of $\varphi_{\mathbf{R}}$.

**Case** $X \subseteq X'$

Given $I(X) = \varphi$ and $I(X') = \psi$, $X \subseteq X'$ holds iff $\varphi \models \psi$. This is the definition of **SE** and thus $\mathbf{R}$ can efficiently verify this case iff it efficiently supports **SE**.

**Case** $\bigcap_{X_i \in \mathcal{X}} X_i \subseteq X'$

In this case we allow a bounded intersection on the left side. Given $I(X_i) = \varphi_i$ and $I(X') = \psi$, we define a set of formulas $\Phi = \{\varphi_i | X_i \in \mathcal{X}\}$.

For the case $|\Phi| = 0$, we do not have any $\varphi_i$, and the statement we want to verify is $\top \models \psi$, which is another way of asking if $\psi$ is valid. This can be verified efficiently iff $\mathbf{R}$ efficiently supports **VA**.

If $|\Phi| = 1$, the statement results in $\varphi \models \psi$, which was covered in the above case.

If $|\Phi| > 1$, the left side consists of a conjunction of formulas. If $\mathbf{R}$ efficiently supports $\land$**BC**, then this conjunction can be transformed into a single formula and we have case $\varphi \models \psi$ again, meaning **SE** and $\land$**BC** are sufficient for efficient verification.

If $\mathbf{R}$ does not efficiently support $\land$**BC**, but **toDNF** and **IM** instead, we can also do the following: Transform all $\varphi_i \in \Phi$ to DNF, then multiply the conjunction of DNFs out, resulting in one DNF. The cubes of this DNF are the implicants which can be tested separately with **IM**. Multiplying out the DNFs is exponential in the number of conjunctions, but polynomial for bounded $|\mathcal{X}|$.

**Case** $X \subseteq \bigcup_{X'_i \in \mathcal{X}'} X'_i$

A similar conclusion can be drawn for the case where we allow a bounded union on the right side. We are given $I(X) = \varphi$, $I(X'_i) = \psi_i$ for each $X'_i \in \mathcal{X}'$ and define $\Psi = \{\psi_i | X'_i \in \mathcal{X}'\}$.

If $|\Psi| = 0$, we need to verify $\varphi \models \bot$, which can be done by testing $\varphi$ on consistency.

The case $|\Psi| = 1$ again reduces to case $X \subseteq X'$.

For $|\Psi| > 1$, we can merge the disjunction in one formula with $\vee$**BC**. Alternatively we can turn each $\psi_i$ into a CNF with **toCNF** and multiply out the CNFs, resulting in one CNF. We then check for each clause if it is entailed by $\varphi$ with **CE**.

#### Case $\bigcap_{X_i \in \mathcal{X}} X_i \subseteq \bigcup_{X'_i \in \mathcal{X}'} X'_i$

If we allow both a bounded intersection on the left side and a bounded union on the right side, we have 4 new cases:

- $|\Phi| = 0$ and $|\Psi| = 0$: This results in $\top \models \bot$ which never holds.
- $|\Phi| = 0$ and $|\Psi| > 1$: In this case we can either build the disjunction of $\Psi$ into one formula with $\vee$**BC** and test validity with **VA**; or transform each $\psi_i \in \Psi$ to CNF with **toCNF** and multiply them out. Testing the resulting CNF on validity can be done easily by testing each clause.
- $|\Phi| > 1$ and $|\Psi| = 0$: Analogous to above we either build the conjunction of $\Phi$ with $\wedge$**BC** and test it on consistency with **CO**; or build DNFs from each $\varphi_i \in \Phi$ with **toDNF**, multiply them out and test the resulting DNF on consistency by testing each cube.
- $|\Phi| > 1$ and $|\Psi| > 1$: If **R** supports both $\wedge$**BC** and $\vee$**BC** efficiently, we can reduce the case to $\varphi \models \psi$.

  If we build a CNF on the right side and want to use **CE** on the left, **R** needs to efficiently support $\wedge$**BC** such that we can build a single formula from the conjunction of $\Phi$, on which we can then test clausal entailment of each clause. Analogously when building a DNF on the left side and using **IM** on the right, we need $\vee$**BC** to build a single formula from the disjunction of $\Psi$.

#### Case $\bigcap_{L_i \in \mathcal{L}} L_i \subseteq \bigcup_{L'_i \in \mathcal{L}'} L'_i$

As a next and final step for basic statement **B1**, we allow complements of state set variables. We are given $I(X_i) = \varphi_i^+$ for all $X_i \in \mathcal{L}$, $I(X_i) = \varphi_i^-$ for all $\overline{X_i} \in \mathcal{L}$, $I(X'_i) = \psi_i^+$ for all $X'_i \in \mathcal{L}'$ and $\psi_i^-$ for all $\overline{X'_i} \in \mathcal{L}'$. We define $\Phi^+$ as the set of all $\varphi_i^+$, $\Phi^-$ as the set of all $\varphi_i^-$, $\Psi^+$ as the set of all $\psi_i^+$ and $\Psi^-$ as the set of all $\psi_i^-$.

The corresponding statement to verify in propositional logic is now:

$$\bigwedge_{\varphi_i^+ \in \Phi^+} \varphi_i^+ \wedge \bigwedge_{\varphi_i^- \in \Phi^-} \neg\varphi_i^- \models \bigvee_{\psi_i^+ \in \Psi^+} \psi_i^+ \vee \bigvee_{\psi_i^- \in \Psi^-} \neg\psi_i^- \tag{5.1}$$

There is a simple way to express an equivalent statement where no negations occur: We have that $\phi \wedge \neg\phi' \models \psi$ iff $\phi \models \psi \vee \phi'$, and $\phi \models \psi \vee \neg\psi'$ iff $\phi \wedge \psi' \models \psi$. This means

above statement holds iff

$$\bigwedge_{\varphi_i^+ \in \Phi^+} \varphi_i^+ \wedge \bigwedge_{\psi_i^- \in \Psi^-} \psi_i^- \models \bigvee_{\psi_i^+ \in \Psi^+} \psi_i^+ \vee \bigvee_{\varphi_i^- \in \Phi^-} \varphi_i^- \tag{5.2}$$

holds. This reduces the case to case $\bigcap_{X_i \in \mathcal{X}} X_i \subseteq \bigcup_{X'_i \in \mathcal{X}'} X'_i$.

The following theorem summarizes the above discussion under which circumstances basic statement **B1** can be verified efficiently:

**Theorem 5.5.** *The statement $\bigcap_{L_i \in \mathcal{L}} L_i \subseteq \bigcup_{L'_i \in \mathcal{L}'} L'_i$ where $|\mathcal{L}| + |\mathcal{L}'| \leq r$ and the involved state set variables are represented with a set of $\mathbf{R}$-formulas $\Phi$ can be verified in polynomial time in $\|\Phi\|$ if $\mathbf{R}$ efficiently supports one of the options in the corresponding cell:*

| | $\mathcal{L}^+ + \mathcal{L}'^- = 0$ | $\mathcal{L}^+ + \mathcal{L}'^- = 1$ | $\mathcal{L}^+ + \mathcal{L}'^- > 1$ |
|---|---|---|---|
| $\mathcal{L}^- + \mathcal{L}'^+ = 0$ | | ***CO*** | ***CO**, $\wedge$**BC***<br>***toDNF*** |
| $\mathcal{L}^- + \mathcal{L}'^+ = 1$ | ***VA*** | ***SE*** | ***SE**, $\wedge$**BC***<br>***toDNF**, **IM*** |
| $\mathcal{L}^- + \mathcal{L}'^+ > 1$ | ***VA**, $\vee$**BC***<br>***toCNF*** | ***SE**, $\vee$**BC***<br>***toCNF**, **CE*** | ***SE**, $\wedge$**BC**, $\vee$**BC***<br>***toDNF**, **IM**, $\vee$**BC***<br>***toCNF**, **CE**, $\wedge$**BC*** |

*where $\mathcal{X}^+$ is the number of non-negated literals in $\mathcal{X}$ and $\mathcal{X}^-$ the number of negated literals in $\mathcal{X}$ for $\mathcal{X} \in \{\mathcal{L}, \mathcal{L}'\}$.*

**Cases** $(\bigcap_{X_i \in \mathcal{X}} X_i)[A] \cap \bigcap_{L_i \in \mathcal{L}} L_i \subseteq \bigcup_{L'_i \in \mathcal{L}'} L'_i$ **/** $[A](\bigcap_{X_i \in \mathcal{X}} X_i) \cap \bigcap_{L_i \in \mathcal{L}} L_i \subseteq \bigcup_{L'_i \in \mathcal{L}'} L'_i$

For basic statements **B2** and **B3**, we allow to add either a progression or regression of a bounded conjunction of state set variables on the left side. In both cases the statement holds iff for all actions $a \in A$ the corresponding statement when replacing $A$ with $a$ holds.

Given action $a$ we define a precondition formula $\rho = \bigwedge_{v_p \in pre(a)} v_p$, an effect formula $\epsilon = \bigwedge_{v_a \in add(a)} v_a \wedge \bigwedge_{v_d \in (del(a) \setminus add(a))} \neg v_d$, and variable set $E = add(a) \cup del(a)$. We also define $\varphi_i^+$, $\varphi_i^-$, $\Phi^+$, $\Phi^-$, $\psi_i^+$, $\psi_i^-$, $\Psi^+$ and $\Psi^-$ as in above case and additionally define $I(X_i) = \varphi_i^\rightarrow$ for each $X_i \in \mathcal{X}$ and $\Phi^\rightarrow$ for the set of all $\varphi_i^\rightarrow$.

By applying Lemma 4.1 to replace the forget operator with a renaming of variables, we now need to verify the following entailments:

$$(\bigwedge_{\varphi_i^\rightarrow \in \Phi^\rightarrow} \varphi_i^\rightarrow \wedge \rho)[E \rightarrow E'] \wedge \epsilon \wedge \bigwedge_{\varphi_i^+ \in \Phi^+} \varphi_i^+ \wedge \bigwedge_{\psi_i^- \in \Psi^-} \psi_i^- \models \bigvee_{\psi_i^+ \in \Psi^+} \psi_i^+ \vee \bigvee_{\varphi_i^- \in \Phi^-} \varphi_i^- \tag{5.3}$$

$$(\bigwedge_{\varphi_i^\rightarrow \in \Phi^\rightarrow} \varphi_i^\rightarrow \wedge \epsilon)[E \rightarrow E'] \wedge \rho \wedge \bigwedge_{\varphi_i^+ \in \Phi^+} \varphi_i^+ \wedge \bigwedge_{\psi_i^- \in \Psi^-} \psi_i^- \models \bigvee_{\psi_i^+ \in \Psi^+} \psi_i^+ \vee \bigvee_{\varphi_i^- \in \Phi^-} \varphi_i^- \tag{5.4}$$

If we choose the renamed variables $E'$ the same way as in the proof of Theorem 4.4, we can represent the progression and regression with $\wedge$**BC**, **CL** and **RN**$_\prec$. The rest of the formula is identical to basic statement **B1**. Since we need $\wedge$**BC** either way, we only distinguish cases based on the amount of formulas on the right side of the entailment. If $|\Psi^+ \cup \Phi^-| = 0$, we additionally only need **CO**. For $|\Psi^+ \cup \Phi^-| = 1$, **CO** gets replaced by **SE**. Finally, if $|\Psi^+ \cup \Phi^-| > 1$ we also need $\vee$**BC**, or transform the right side to a CNF with **toCNF** and use **CE** instead of **SE**.

**Theorem 5.6.** *The statements* $(\bigcap_{X_i \in \mathcal{X}} X_i)[A] \cap \bigcap_{L_i \in \mathcal{L}} \subseteq \bigcup_{L'_i \in \mathcal{L}'} L'_i$ *and* $[A](\bigcap_{X_i \in \mathcal{X}} X_i) \cap \bigcap_{L_i \in \mathcal{L}} \subseteq \bigcup_{L'_i \in \mathcal{L}'} L'_i$ *where* $|\mathcal{X}| + |\mathcal{L}| + |\mathcal{L}'| \leq r$ *and the involved state set variables are represented with a set of* $\mathbf{R}$*-formulas* $\Phi$ *can be verified in time polynomial in* $\|\Phi\|$ *and* $|A|$ *if* $\mathbf{R}$ *efficiently supports one of the options in the corresponding cell:*

| | |
|---|---|
| $\mathcal{L}^- + \mathcal{L}'^+ = 0$ | ***CO**, $\wedge$**BC**, **CL**, **RN**$_\prec$* |
| $\mathcal{L}^- + \mathcal{L}'^+ = 1$ | ***SE**, $\wedge$**BC**, **CL**, **RN**$_\prec$* |
| $\mathcal{L}^- + \mathcal{L}'^+ > 1$ | ***SE**, $\vee$**BC**, $\wedge$**BC**, **CL**, **RN**$_\prec$*<br>***toCNF**, **CE**, $\wedge$**BC**, **CL**, **RN**$_\prec$* |

*where* $\mathcal{X}^+$ *is the number of non-negated literals in* $\mathcal{X}$ *and* $\mathcal{X}^-$ *the number of negated literals in* $\mathcal{X}$ *for* $\mathcal{X} \in \{\mathcal{L}, \mathcal{L}'\}$*.*

### 5.3.2 Mixed Representations

In order to bridge representation formalisms we allow the basic statement **B4** $L \subseteq L'$ to use different formalisms $\mathbf{R}$ and $\mathbf{R}'$ to represent the state set variable underlying $L$ and $L'$. Given that both $L$ and $L'$ can either be state set variables or their complement, the statement can be split up in four cases: $X \subseteq X'$, $\overline{X} \subseteq X'$, $X \subseteq \overline{X'}$ and $\overline{X} \subseteq \overline{X'}$. Given $I(X) = \varphi_\mathbf{R}$ and $I(X') = \psi_{\mathbf{R}'}$ we analyze the equivalent propositional logic statements:

#### Case $\varphi_\mathbf{R} \models \psi_{\mathbf{R}'}$

We first consider the basic case with no complement. Since $\varphi_\mathbf{R}$ and $\psi_{\mathbf{R}'}$ use different formalisms we cannot use **SE** directly. There are however two approaches we used in the single representation case that can also be used here: **toDNF** and **IM**, and **toCNF** and **CE**. In the single representation case we used this when a formalism does not efficiently support $\wedge$**BC** or $\vee$**BC** respectively, but the general idea of transforming one side to DNF or CNF and using **IM** or **CE** on the other side can also be applied when different formalisms are involved.

If $\mathbf{R}$ efficiently supports **toDNF**, we can transform $\varphi_\mathbf{R}$ to a DNF and then for each cube check if it implies $\psi_{\mathbf{R}'}$, as long as $\mathbf{R}'$ supports **IM**. Similarly, if $\mathbf{R}'$ efficiently supports **toCNF** and $\mathbf{R}$ efficiently supports **CE**, we transform $\psi_{\mathbf{R}'}$ to CNF and check for each clause if it is entailed by $\varphi_\mathbf{R}$.

The idea of breaking down a formula into clauses or cubes can be taken even further by breaking the formula down to its models. If we can enumerate all models for $\varphi_{\mathbf{R}}$, then we can check for each model if it is also a model for $\psi_{\mathbf{R}'}$. This requires **ME** for $\mathbf{R}$ and **MO** for $\mathbf{R}'$. However, since **ME** is polynomial in the size of the formula *and the amount of models*, either $\mathbf{R}$ or $\mathbf{R}'$ must additionally be non-succinct[1]. If $\varphi_{\mathbf{R}}$ is non-succinct, then enumerating its models is polynomial in $\|\varphi_{\mathbf{R}}\|$. If $\psi_{\mathbf{R}'}$ is non-succinct, then we need to enumerate at most $\|\psi_{\mathbf{R}'}\| + 1$ models of $\varphi_{\mathbf{R}}$, at which point at least one model is not contained in $\psi_{\mathbf{R}'}$.

The above argument only holds if $\varphi_{\mathbf{R}}$ and $\psi_{\mathbf{R}'}$ mention the same set of variables. For cases where the mentioned variables differ, we need to impose more restrictions:

1. $\mathit{vars}(\varphi_{\mathbf{R}}) \setminus \mathit{vars}(\psi_{\mathbf{R}'}) \neq \emptyset$: We can forget the variables for each model of $\varphi_{\mathbf{R}}$ and test **MO** in $\psi_{\mathbf{R}'}$ with the reduced model. This is only polynomial in the representation size if $\varphi_{\mathbf{R}}$ *is non-succinct*. Otherwise, we might enumerate an amount of models that is exponential in $\|\varphi_{\mathbf{R}}\|$ and reduce all of them to the same model for $\psi_{\mathbf{R}'}$. As an example, consider formulas $\chi_1 = x_1 \wedge (x_2 \vee \cdots \vee x_n)$ and $\chi_2 = x_1$. If we represent $\chi_1$ as BDD, its size is linear in $n$, and an explicit enumeration of $\chi_2$ over variable set $V = \{x_1\}$ has size linear in its variables. But $\chi_1$ has $2^{(n-1)} - 1$ models, thus enumerating them is not polynomial in either representation size.

2. $\mathit{vars}(\psi_{\mathbf{R}'}) \setminus \mathit{vars}(\varphi_{\mathbf{R}}) \neq \emptyset$: We expand the models of $\varphi_{\mathbf{R}}$ to the variables occurring in $\psi_{\mathbf{R}'}$. This step is exponential in the amount of variables added; but if $\psi_{\mathbf{R}'}$ *is non-succinct* we are still guaranteed to enumerate at most $\|\psi_{\mathbf{R}'}\|$ "extended" models (since after this we must have enumerated at least one non-model of $\psi_{\mathbf{R}'}$).

**Case** $\neg\varphi_{\mathbf{R}} \models \psi_{\mathbf{R}'}$

If $\mathbf{R}$ efficiently supports $\neg$**C** we can reduce the case to $\varphi_{\mathbf{R}} \models \psi_{\mathbf{R}'}$. Otherwise we observe that instead of transforming $\varphi_{\mathbf{R}}$ to a DNF we can use **toCNF** and transform it to a CNF. From this we can efficiently construct a DNF representing $\neg\varphi_{\mathbf{R}}$ with de Morgan's law.

Using **ME** and **MO** is not directly possible anymore since we cannot efficiently enumerate the non-models of $\varphi_{\mathbf{R}}$ needed in this case. But if $\mathbf{R}$ efficiently supports **MO** and **CT**, and $\mathbf{R}'$ efficiently supports **ME** and is non-succinct, we can instead count those interpretations that are models of $\psi_{\mathbf{R}'}$ but not of $\varphi_{\mathbf{R}}$ by enumerating all models of $\psi_{\mathbf{R}'}$ and counting them only if they are not models of $\varphi_{\mathbf{R}}$ (which can be done with **MO**). This must be equal to the number of models of $\neg\varphi_{\mathbf{R}}$, which we can compute with **CT** since $|\neg\varphi_{\mathbf{R}}| = 2^{\mathit{vars}(\varphi_{\mathbf{R}})} - |\varphi_{\mathbf{R}}|$. We must however again pay attention if the two formulas involve different variables. We can only support the case $\mathit{vars}(\varphi_{\mathbf{R}}) \subseteq \mathit{vars}(\psi_{\mathbf{R}'})$, where we forget the variables not occurring in $\varphi_{\mathbf{R}}$ for each model of $\psi_{\mathbf{R}'}$. Even if $\varphi_{\mathbf{R}}$ is not succinct as well we could not guarantee non-exponential overhead if $\varphi_{\mathbf{R}}$ mentions variables not in $\psi_{\mathbf{R}'}$ because we might need $|\neg\varphi_{\mathbf{R}}|$ **MO** checks, which can be exponential in $|\varphi_{\mathbf{R}}|$.

[1]Non-succinct in this context means the representation size of the formula grows linearly in the amount of models.

Finally we observe that the statement $\neg\varphi_{\mathbf{R}} \models \psi_{\mathbf{R}'}$ is equivalent to $\neg\psi_{\mathbf{R}'} \models \varphi_{\mathbf{R}}$, meaning that the roles of $\mathbf{R}$ and $\mathbf{R}'$ can be exchanged.

**Case** $\varphi_{\mathbf{R}} \models \neg\psi_{\mathbf{R}'}$

As in the case above this case can be reduced to the first one if $\mathbf{R}'$ efficiently supports $\neg$**C**. Otherwise we can build a DNF from $\psi_{\mathbf{R}'}$ if $\mathbf{R}'$ efficiently supports **toDNF**. Negating this DNF with de Morgan results in a CNF representing $\neg\psi_{\mathbf{R}'}$, and we can check for each clause if it is entailed by $\varphi_{\mathbf{R}}$ if $\mathbf{R}$ efficiently supports **CE**.

The approach with **ME** and **MO** is similar to the first case, only this time $\varphi_{\mathbf{R}}$ must be non-succinct and $vars(\psi_{\mathbf{R}'}) \subseteq vars(\varphi_{\mathbf{R}})$ must hold, since the negation of a non-succinct formula renders the non-succinctness useless for our approach.

As with the previous case we see that $\varphi_{\mathbf{R}} \models \neg\psi_{\mathbf{R}'}$ is equivalent to $\psi_{\mathbf{R}'} \models \neg\varphi_{\mathbf{R}}$, thus the roles of $\mathbf{R}$ and $\mathbf{R}'$ can be exchanged again.

**Case** $\neg\varphi_{\mathbf{R}} \models \neg\psi_{\mathbf{R}'}$

This statement is equivalent to $\psi_{\mathbf{R}'} \models \varphi_{\mathbf{R}}$, and can thus be reduced to the first case by exchanging $\mathbf{R}$ and $\mathbf{R}'$, even without requiring $\neg$**C** from either formalism.

**Theorem 5.7.** *The statement $L \subseteq L'$ where the two involved state set variables are represented by $\varphi_{\mathbf{R}}$ and $\psi_{\mathbf{R}'}$ with $\mathbf{R} \neq \mathbf{R}'$ can be verified in polynomial time in $\|\varphi_{\mathbf{R}}\|$ and $\|\psi_{\mathbf{R}'}\|$ in the following cases:*

| | $\mathbf{R}$ | $\mathbf{R}'$ |
|---|---|---|
| | ***ME****, ns* | ***MO*** |
| $\varphi_{\mathbf{R}} \models \psi_{\mathbf{R}'}$ | ***toDNF*** | ***IM*** |
| $\neg\psi_{\mathbf{R}'} \models \neg\varphi_{\mathbf{R}}$ | ***CE*** | ***toCNF*** |
| | ***ME*** | ***MO****, ns* |
| | ***ME****, ns* | ***MO****,* ***CT*** |
| $\neg\varphi_{\mathbf{R}} \models \psi_{\mathbf{R}'}$ | ***toCNF*** | ***IM*** |
| $\neg\psi_{\mathbf{R}'} \models \varphi_{\mathbf{R}}$ | ***IM*** | ***toCNF*** |
| | ***MO****,* ***CT*** | ***ME****, ns* |
| | ***ME****, ns* | ***MO*** |
| $\varphi_{\mathbf{R}} \models \neg\psi_{\mathbf{R}'}$ | ***toDNF*** | ***CE*** |
| $\psi_{\mathbf{R}'} \models \neg\varphi_{\mathbf{R}}$ | ***CE*** | ***toDNF*** |
| | ***MO*** | ***ME****, ns* |

*where "ns" means that the formalism is non-succinct (i.e. the representation size of the formula is in best case linear in the amount of models). If the other involved formula is succinct, it cannot contain variables not mentioned in the non-succinct formula.*

*If $\mathbf{R}$ ($\mathbf{R}'$) supports $\neg$**C** and $\neg\varphi_{\mathbf{R}}$ ($\neg\psi_{\mathbf{R}'}$) occurs, we can also reduce the case to $\varphi_{\mathbf{R}} \models \psi_{\mathbf{R}'}$.*

Looking at our concrete formalisms, we see that any statement where MODS is involved can be verified efficiently. In most cases, we utilize the non-succinctness of the formula, since the other formalism then only needs to support **MO** (possibly in combination with **CT**) or **ME**. All formalisms support **MO** and **ME**, but Horn and 2CNF do no support **CT**, which is needed for $\neg\varphi_{\mathbf{R}} \models \psi_{\mathbf{R}'}$. In these cases we can however use **toCNF** and **IM**.

Since BDDs, Horn and 2CNF all are succinct, we need to rely on **toDNF** and **toCNF** in those cases. However, Horn and 2CNF do not support **toDNF** and BDD supports neither. Thus, all statements of type $\varphi_{\mathbf{R}} \models \neg\psi_{\mathbf{R}'}$ involving two of those three formalisms cannot be verified efficiently since they require **toDNF** from one of the two formalisms. Furthermore, $\varphi_{\text{Horn/2CNF}} \models \psi_{\text{BDD}}$ and the equivalent $\neg\psi_{\text{BDD}} \models \neg\varphi_{\text{Horn/2CNF}}$ cannot be verified efficiently either since we would need either **toDNF** from Horn/2CNF or **toCNF** from BDD. All other cases are however efficiently supported.

## 5.4 Comparison to Inductive Certificates

All unsolvability certificates in our previous approach are based on *inductive sets* (i.e. sets $S$ with $S[A^\Pi] \subseteq S$) that contain the initial state and no goal state. On a high level, such certificates can be translated into the proof system the following way: We first show that $S$ is dead with rule **PG** (instantiating $S'$ with $\emptyset$). Then, since $I^\Pi \in S$, we can show that $\{I^\Pi\}$ is dead with **SD** and can conclude the proof with **CI**.

For the simplest form of *inductive certificates*, the translation is very straightforward:

**Theorem 5.8.** *Given an inductive $\mathbf{R}$-certificate $S$ for $\Pi$ (meaning $I^\Pi \in S$, $S \cap S_G^\Pi = \emptyset$ and $S$ is inductive), we can construct a proof in the proof system in linear time. Verifying the proof can be done with the same operations that are needed for inductive certificates, namely **MO**, **CE**, **SE**, $\wedge$**BC**, **CL** and **RN**$_\prec$.*

**Proof:** Let $S$ be an inductive certificate for $\Pi$. We start with **ED** deducing (1) $\emptyset$ is dead. Since $S$ contains no goal states, we state (2) $(S \cap S_G^\Pi) \sqsubseteq \emptyset$ with **B1** and apply **SD** with (1) and (2) to get (3) $(S \cap S_G^\Pi)$ is dead. Further, since $S$ is inductive we can state (4) $S[A^\Pi] \sqsubseteq S$ with **B2**. We apply **UR** to deduce (5) $S \sqsubseteq (S \cup \emptyset)$ and use **ST** with (4) and (5), resulting in (6) $S[A^\Pi] \sqsubseteq (S \cup \emptyset)$.[2] Using **PG** with (6), (1) and (3), we conclude (7) $S$ is dead. Since $I^\Pi \in S$, we can state (8) $\{I^\Pi\} \sqsubseteq S$ with **B1**, and apply **SD** with (7) and (8) to obtain (9) $\{I^\Pi\}$ is dead. We conclude the proof with **CI** applied to (9) resulting in (10) "unsolvable".

We only require operations from $\mathbf{R}$ for representing constant formulas (requiring **CL**) and verifying the basic statements occurring at (2), (4) and (8). For (4), we need **SE**, $\wedge$**BC**, **CL** and **RN**$_\prec$, and with this (2) and (8) is already covered as well.

[2] We need judgment $S[A^\Pi] \sqsubseteq (S \cup \emptyset)$ rather than $S[A^\Pi] \subseteq S$ for **PG**. We could also directly state the needed judgment but then $\mathbf{R}$ theoretically would need to efficiently support $\vee$**BC** according to Theorem 5.6, even if practically this is not needed since the second set is $\emptyset$.

We technically do not even require **MO** and **CE** here. However, any formalism efficiently supporting **SE**, $\wedge$**BC** and **CL** can also handle **MO** and **CE** For **MO**, we can build a $\mathbf{R}$-formula representing the model with **CL**, and test if the model is contained with **SE**. For **CE**, we build the negation of $\gamma$ with **CL** and test $\varphi \wedge \neg\gamma \models \bot$ with **SE**. □

Since inductive certificates are complete, this also directly means that our proof system is complete with respect to judgment "task unsolvable":

**Theorem 5.9** (soundness and completeness of the proof system)**.** *Given a STRIPS planning tasks* $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$*, there is an proof in the proof system for* $\Pi$ *iff* $\Pi$ *is unsolvable.*

**Proof:** Soundness follows from the correctness of all inference rules and basic statements. Completeness (with respect to judgment "task unsolvable") follows from the fact that we can for any unsolvable task $\Pi$ translate an inductive certificate (which must exist since inductive certificates are complete) into a proof in the proof system. □

For $r$-disjunctive and $r$-conjunctive certificates the translation becomes more involved, since $S$ is represented as a union or intersection of a family $\mathcal{F}$ of state sets and it is assumed that the explicit union or intersection cannot be represented efficiently. Instead we need to express the reasoning in the proofs of Theorem 4.5 and 4.7 within the proof system. Specifically, we need to show that subset statements such as $\bigcup_{S'_i \in \mathcal{F}'} S'_i \subseteq \bigcup_{S_j \in \mathcal{F}} S_j$ (where $\mathcal{F}' \subseteq \mathcal{F}$) hold. While this is possible with the set theory rules we have included in the proof system, building the proof can no longer be done in linear time.

In order to avoid different binary representations of a union/intersection of several (indexed) sets, we build the binary representation in a linear ascending fashion. Given a family of state sets $\mathcal{F} = \{S_i, S_j, \dots, S_k\}$, we denote the union as $\mathcal{F}_\cup = (\dots(S_i \cup S_j) \dots \cup S_k)$ and the intersection as $\mathcal{F}_\cap = (\dots(S_i \cap S_j) \dots \cap S_k)$, where $i < j < k$.

**$r$-disjunctive Certificates** An $r$-disjunctive certificate is a family of state sets $\mathcal{F} = \{S_1, \dots S_{|\mathcal{F}|}\}$, such that (a) $I^\Pi \in S_k$ for some $S_k \in \mathcal{F}$, (b) $S_i \cap S_G^\Pi = \emptyset$ for all $S_i \in \mathcal{F}$ and (c) $S_i[a] \subseteq \bigcup_{S'_j \in \mathcal{F}'} S'_j$ for all $S_i \in \mathcal{F}$, $a \in A^\Pi$ where $\mathcal{F}'$ is a subset of $\mathcal{F}$ of at most size $r$. All these properties can be expressed as basic statements for the proof (for (b) we have a subset instead of an equality relation). We then need to show within the proof that from (b) follows $(\mathcal{F}_\cup \cap S_G^\Pi) \sqsubseteq \emptyset$, and from (c) follows $\mathcal{F}_\cup[A^\Pi] \sqsubseteq (\mathcal{F}_\cup \cup \emptyset)$. With this we can show that $\mathcal{F}_\cup$ is dead, from which we conclude with (a) and $S_i \sqsubseteq \mathcal{F}_\cup$ that $\{I^\Pi\}$ is dead and thus the task is unsolvable.

**Theorem 5.10.** *Given an $r$-disjunctive certificate* $\mathcal{F} = \{S_1, \dots, S_n\}$ *for* $\Pi$*, we can construct a proof in the proof system in time* $O(|\mathcal{F}|^2 + |A^\Pi| \cdot |\mathcal{F}| \cdot r)$ *showing that* $\Pi$ *is unsolvable. Verifying the proof can be done with the same operations that are needed for $r$-disjunctive certificates, namely **MO**, **CE**, $\wedge$**BC**, **CL**, **RN**$_\prec$ and either (i) $\vee$**BC** and **SE** or (ii) **toCNF**.*

**Proof:** We express (a) as basic statement **B1**: $\{I^\Pi\} \sqsubseteq S_k$, (b) as $|\mathcal{F}|$ basic statements **B1**: $S_i \cap S_G^\Pi \sqsubseteq \emptyset$, and (c) as $|A^\Pi| \cdot |\mathcal{F}|$ basic statements **B2** $S_i[a_j] \sqsubseteq \mathcal{F}_\cup^{i,j}$, where $\mathcal{F}_\cup^{i,j}$ represents the corresponding subfamily for this particular set and action. Furthermore, we establish basic statement **B5**: $A^\Pi \sqsubseteq (\dots(a_1 \cup a_2) \dots \cup a_{|A^\Pi|})$ and that $\emptyset$ is dead with **ED**.

We first want to show that $\mathcal{F}_\cup[A^\Pi] \sqsubseteq \mathcal{F}_\cup$ holds. As a preparation for this we first show that $S_i \sqsubseteq \mathcal{F}_\cup$ holds for all $S_i \in \mathcal{F}$. This requires $O(|\mathcal{F}|^2)$ steps. For each $S_i$, we first apply **UL** to show that $S_i$ is a subset of the union of all sets up to $S_i$, and then extend the union one set at a time with **UD** and **ST**, requiring $|\mathcal{F}| - i$ extension steps:

| # | judgment | rule | premises |
|---|---|---|---|
| $(i)$ | $S_i \sqsubseteq (\dots(S_1 \cup S_2) \dots \cup S_i)$ | **UL** | |
| $(i+1)$ | $(\dots(S_1 \cup S_2) \dots \cup S_i) \sqsubseteq ((\dots(S_1 \cup S_2) \dots \cup S_i) \cup S_{i+1})$ | **UR** | |
| $(i+2)$ | $S_i \sqsubseteq ((\dots(S_1 \cup S_2) \dots \cup S_i) \cup S_{i+1})$ | **ST** | $(i)$,$(i+1)$ |

Next, we can show for each $S_i \in \mathcal{F}$, $a \in A^\Pi$ that $S_i[a] \sqsubseteq \mathcal{F}_\cup$ holds by applying **ST** to basic statement $S_i[a_j] \sqsubseteq \mathcal{F}_\cup^{i,j}$ and judgment $\mathcal{F}_\cup^{i,j} \sqsubseteq \mathcal{F}_\cup$. The latter can be obtained by using **SU** $r-1$ times with the already derived judgments $S_i \sqsubseteq \mathcal{F}_\cup$. In total this part requires $O(|\mathcal{F}| \cdot |A^\Pi| \cdot r)$ steps.

| # | judgment | rule | premises |
|---|---|---|---|
| $(x)$ | $S_x \sqsubseteq \mathcal{F}_\cup$ | | |
| $(y)$ | $S_y \sqsubseteq \mathcal{F}_\cup$ | | |
| $(z)$ | $S_z \sqsubseteq \mathcal{F}_\cup$ | | |
| $(i)$ | $S_i[a_j] \sqsubseteq \mathcal{F}_\cup^{i,j}$ | **B2** | |
| $(i+1)$ | $(S_x \cup S_y) \sqsubseteq \mathcal{F}_\cup$ | **SU** | $(x)$,$(y)$ |
| $\dots$ | | | |
| $(i+r-1)$ | $\mathcal{F}_\cup^{i,j} \sqsubseteq \mathcal{F}_\cup$ | **SU** | $(i+r-2)$,$(z)$ |
| $(i+r)$ | $S_i[a_j] \sqsubseteq \mathcal{F}_\cup$ | **ST** | $(i)$,$(i+r-1)$ |

From these judgments we can now for each $S_i \in \mathcal{F}$ first derive that $S_i[A^\Pi] \sqsubseteq \mathcal{F}_\cup$ holds with **AU** and **AT**, and combine this together with **PU** to the judgment $\mathcal{F}_\cup[A^\Pi] \sqsubseteq \mathcal{F}_\cup$, requiring $O(|\mathcal{F}| \cdot |A^\Pi|)$ steps. Extending the judgment to the required (A) $\mathcal{F}_\cup[A^\Pi] \sqsubseteq \mathcal{F}_\cup \cup \emptyset$ is then simply achieved with **UR** ($\mathcal{F}_\cup \sqsubseteq (\mathcal{F}_\cup \cup \emptyset)$) and **ST**.

To show that $(\mathcal{F}_\cup \cap S_G^\Pi) \sqsubseteq \emptyset$ holds, we need basic statement **B1**: $(S_i \cap S_G^\Pi) \sqsubseteq \emptyset$ for all $S_i \in \mathcal{F}$, and combine them together with help of **SU**, **DI** and **ST**, requiring $O(|\mathcal{F}|)$ steps.

| # | judgment | rule | premises |
|---|---|---|---|
| $(x)$ | $((\dots) \cap S_G^\Pi) \sqsubseteq \emptyset$ | | |
| $(i)$ | $(S_i \cap S_G^\Pi) \sqsubseteq \emptyset$ | **B1** | |
| $(i+1)$ | $(((\dots) \cap S_G^\Pi) \cup (S_i \cap S_G^\Pi)) \sqsubseteq \emptyset$ | **SU** | $(x)$,$(i)$ |
| $(i+2)$ | $(((...) \cup S_i) \cap S_G^\Pi) \sqsubseteq (((..) \cap S_G^\Pi) \cup (S_i \cap S_G^\Pi))$ | **DI** | |
| $(i+3)$ | $(((\dots) \cup S_i) \cap S_G^\Pi) \sqsubseteq \emptyset$ | **ST** | $(i+2)$,$(i+1)$ |

With this we can conclude that (B) $(\mathcal{F}_\cup \cap S_G^\Pi)$ is dead with the help of **ED** and **SD**. Now we can show with **PG** applied to (A), $\emptyset$ dead and (B) that $\mathcal{F}_\cup$ is dead, and then apply **ST** to basic statement $\{I^\Pi\} \sqsubseteq S_k$ and judgment $S_k \sqsubseteq \mathcal{F}_\cup$ to show that $\{I^\Pi\}$ is in $\mathcal{F}_\cup$, from which follows with **SD** that $\{I^\Pi\}$ is dead and thus the task is unsolvable (**CI**).

Looking at the required operations, the basic statement from (a) requires **CL** and **SE**, for (c) we need $\wedge$**BC**, **CL**, **RN**$_\prec$ and either (i) $\vee$**BC** and **SE**, or (ii) **toCNF** and **CE**, and (b) is then covered with **CL**, **SE** and $\wedge$**BC**. While $r$-disjunctive certificates do not list **SE** as a requirement when using **toCNF**, they are covered through **toCNF** and **CE**: $\varphi \models \psi$ can be checked by transforming $\psi$ to a CNF and checking for each clause $\gamma$ that $\varphi \models \gamma$ holds with **CE**. $\square$

**$r$-conjunctive Certificates** An $r$-conjunctive certificate is a family of state sets $\mathcal{F} = \{S_1, \dots S_{|\mathcal{F}|}\}$, such that (a) $I^\Pi \in S_i$ for all $S_i \in \mathcal{F}$, (b) $(\bigcap_{S_i \in \mathcal{F}^G} S_i) \cap S_G^\Pi = \emptyset$ for a subset $\mathcal{F}^G$ of $\mathcal{F}$ of at most size $r$, and (c) $(\bigcap_{S'_j \in \mathcal{F}'} S'_j)[a] \subseteq S_i$ for all $S_i \in \mathcal{F}$, $a \in A^\Pi$ where $\mathcal{F}'$ is a subset of $\mathcal{F}$ of at most size $r$.

All these properties can be expressed as basic statements for the proof (for (b) we have a subset instead of an equality relation). We then need to show within the proof that from (b) follows $(\mathcal{F}_\cap \cap S_G^\Pi) \sqsubseteq \emptyset)$, and from (c) follows $\mathcal{F}_\cap[A^\Pi] \sqsubseteq (\mathcal{F}_\cap \cup \emptyset)$. With this we can show that $\mathcal{F}_\cap$ is dead. From (a) we can show that $\{I^\Pi\} \sqsubseteq \mathcal{F}_\cap$ holds, from which we conclude that $\{I^\Pi\}$ is dead and the task is unsolvable.

**Theorem 5.11.** *Given an $r$-conjunctive certificate $\mathcal{F} = \{S_1, \dots, S_n\}$ for $\Pi$, we can construct a proof in the proof system in time $O(|\mathcal{F}|^2 + |A^\Pi| \cdot |\mathcal{F}| \cdot r)$ showing that $\Pi$ is unsolvable. Verifying the proof can be done with the same operations that are needed for $r$-conjunctive certificates, namely **MO**, **CE**, **SE**, $\wedge$**BC**, **CL** and **RN**$_\prec$.*

**Proof:** The proof is in many parts similar to the $r$-disjunctive case. We express (a) with $|\mathcal{F}|$ basic statements **B1**: $\{I^\Pi\} \sqsubseteq S_i$ for each $S_i$, (b) with basic statement **B1**: $\mathcal{F}_\cap^G \cap S_G^\Pi \sqsubseteq \emptyset$ and (c) with $|A^\Pi| \cdot |\mathcal{F}|$ basic statements **B2**: $\mathcal{F}_\cap^{i,j}[a_j] \sqsubseteq S_i$, where $\mathcal{F}_\cap^{i,j}$ represents the corresponding subfamily for this particular set and action. Analogously to the disjunctive case we establish the set of all actions and that $\emptyset$ is dead with **B5** and **ED**.

To show that $\mathcal{F}_\cap[A^\Pi] \sqsubseteq \mathcal{F}_\cap$ holds, we first show in $O(|\mathcal{F}|^2)$ steps that $\mathcal{F}_\cap \sqsubseteq S_i$ holds for each $S_i$. For this we first apply **IL** to get that the intersection of all sets up to $S_i$ is a subset of $S_i$, and then add sets $S_j$ with $j > i$ one step at a time with **IR** and **ST**:

| # | judgment | rule | premises |
|---|---|---|---|
| $(i)$ | $(\dots(S_1 \cap S_2) \dots \cap S_i) \sqsubseteq S_i$ | **IL** | |
| $(i+1)$ | $((\dots(S_1 \cap S_2) \dots \cap S_i) \cap S_{i+1}) \sqsubseteq (\dots(S_1 \cap S_2) \dots \cap S_i)$ | **IR** | |
| $(i+2)$ | $((\dots(S_1 \cap S_2) \dots \cap S_i) \cap S_{i+1}) \sqsubseteq S_i$ | **ST** | $(i+1)$,$(i)$ |

Next, we show for each set $S_i$ and action $a$ that $\mathcal{F}_\cap[a] \sqsubseteq S_i$ holds with the help of basic statements from (c) and rules **SI** and **PT**, requiring $O(|\mathcal{F}| \cdot |A^\Pi| \cdot r)$ steps in total:

| # | judgment | rule | premises |
|---|---|---|---|
| $(x)$ | $\mathcal{F}_\cap \sqsubseteq S_x$ | | |
| $(y)$ | $\mathcal{F}_\cap \sqsubseteq S_y$ | | |
| $(z)$ | $\mathcal{F}_\cap \sqsubseteq S_z$ | | |
| $(i)$ | $\mathcal{F}_\cap^{i,j}[a_j] \sqsubseteq S_i$ | **B2** | |
| $(i+1)$ | $\mathcal{F}_\cap \sqsubseteq (S_x \cap S_y)$ | **SI** | $(x)$,$(y)$ |
| ... | | | |
| $(i{+}r{-}1)$ | $\mathcal{F}_\cap \sqsubseteq \mathcal{F}_\cap^{i,j}$ | **SI** | $(i{+}r{-}2)$,$(z)$ |
| $(i+r)$ | $\mathcal{F}_\cap[a_j] \sqsubseteq S_i$ | **PT** | $(i)$,$(i{+}r-1)$ |

Analogously to the disjunctive case we can from these judgments now derive $\mathcal{F}_\cap[A^\Pi] \sqsubseteq S_i$ for each $S_i$ with $O(|A^\Pi|)$ applications of **AU** and one application of **AT**. To deduce $\mathcal{F}_\cap[A^\Pi] \sqsubseteq \mathcal{F}_\cap$ then requires $O(|\mathcal{F}|)$ applications of **SI**. Finally, we derive judgment (A) $\mathcal{F}_\cap[A^\Pi] \sqsubseteq \mathcal{F}_\cap[A^\Pi] \cup \emptyset$ with **UR** and **ST**. In total this part requires $O(|\mathcal{F}| \cdot |A^\Pi|)$ steps.

Showing that (B) $\mathcal{F}_\cap \cap S_G^\Pi$ is dead is easier than in the disjunctive case. We first show $\mathcal{F}_\cap \sqsubseteq \mathcal{F}_\cap^G$ with $r-1$ applications of **SI** (analogous to showing above that $\mathcal{F}_\cap \sqsubseteq \mathcal{F}_\cap^{i.j}$ holds). (B) is then derived with the help of basic statement $\mathcal{F}_\cap^G \cap S_G^\Pi \subseteq \emptyset$ in the following way:

| # | judgment | rule | premises |
|---|---|---|---|
| $(x)$ | $\emptyset$ dead | | |
| $(i-1)$ | $\mathcal{F}_\cap \sqsubseteq \mathcal{F}_\cap^G$ | | |
| $(i)$ | $(\mathcal{F}_\cap \cap S_G^\Pi) \sqsubseteq \mathcal{F}_\cap$ | **IR** | |
| $(i+1)$ | $(\mathcal{F}_\cap \cap S_G^\Pi) \sqsubseteq \mathcal{F}_\cap^G$ | **ST** | $(i)$,$(i-1)$ |
| $(i+2)$ | $(\mathcal{F}_\cap \cap S_G^\Pi) \sqsubseteq S_G^\Pi$ | **IL** | |
| $(i+3)$ | $(\mathcal{F}_\cap \cap S_G^\Pi) \sqsubseteq (\mathcal{F}_\cap^G \cap S_G^\Pi)$ | **SI** | $(i{+}1)$,$(i{+}2)$ |
| $(i+4)$ | $(\mathcal{F}_\cap^G \cap S_G^\Pi) \sqsubseteq \emptyset$ | **B1** | |
| $(i+5)$ | $(\mathcal{F}_\cap \cap S_G^\Pi) \sqsubseteq \emptyset$ | **ST** | $(i{+}3)$,$(i{+}4)$ |
| $(i+6)$ | $(\mathcal{F}_\cap \cap S_G^\Pi)$ dead | **SD** | $(x)$,$(i+5)$ |

We can now derive that $\mathcal{F}_\cap$ is dead by using **PG** with judgments (A), $\emptyset$ dead and (B). It is left to show that $\{I^\Pi\}$ is dead (and thus the task unsolvable). This can be done with the help of basic statements $\{I^\Pi\} \sqsubseteq S_i$ for all $S_i$ and $O(|\mathcal{F}|)$ applications of **SI** to obtain $\{I^\Pi\} \sqsubseteq \mathcal{F}_\cap$, and applying **SD** to show $\{I^\Pi\}$is dead. Rule **CI** concludes the proof.

Looking at the operations needed to verify the proof, basic statements from (a) require **CL** and **SE**, for (c) we need **SE**, $\wedge$**BC**, **CL** and **RN**$_\prec$, and (b) is then covered with **CL**, $\wedge$**BC** and **SE**. □

For both $r$-disjunctive and $r$-conjunctive certificate, the full schematic translation into a proof as well as a concrete example are shown in Appendix A.2.

In summary, our theoretical comparison has shown that all types of inductive certificates can be translated into proofs in our proof system and can be verified efficiently if this is the case for the inductive certificate. However, depending on the certificate type, these proofs can be substantially bigger, because much of the reasoning inherent in the inductive certificate has to be made explicit in the proof. There is an argument to be

made in favor of as fine-grained reasoning as possible for reducing errors, since proving correctness of an inference rule in the proof system is not as involved as proving the entire concept of for example $r$-disjunctive certificates correct. However, the disadvantage of making the reasoning explicit is that verifying the proof will require more time than verifying the properties of the inductive certificate.

# Part II

# Applications

# 6 Search Algorithms

Most state-of-the-art planning systems utilize some kind of search procedure on the induced transition graph of the task $\Pi$ in order to find plans. A general graph search starts from one state (or a set of states) and iteratively discovers new states by *expanding* previously seen states. The states that should be expanded are kept in an *open list* and in each iteration states are taken out in order to be expanded. To avoid re-expanding the same states most search algorithms also keep a *closed list*, where expanded states are remembered, and only expand a state if it is not in the closed list yet.

The three most commonly used search algorithm groups are *depth-first*, *breadth-first* and *best-first search*. Depth-first search expands states in the open list in a FIFO order, meaning the most recently discovered state is expanded first. Breadth-first search on the other hand expands state in a LIFO order, meaning it expands the search space in "layers": first all states immediately adjacent to the starting point are expanded, then the states adjacent to those and so forth. Finally, best-first-search assigns each state in the open list some priority and expands states accordingly. Uniform cost search prioritizes states based on the cost from the starting point to them, thus states nearer to the starting point are expanded first. This guarantees that the first solution found is *optimal*. The other prominent best-first search technique is *heuristic search*, where the expansion priority is determined by a *heuristic* (which estimates the cost to the nearest goal), possibly in combination with other measures like cost from starting point to current state.

*Pruning techniques* are a powerful tool to reduce the part of the search space that must be explored. One common application of pruning is when a heuristic determines that the goal cannot be reached from the current state; in this case it is safe to simply not expand the state since we cannot lose possible solutions.

Proving unsolvability in graph search is essentially achieved by exploring all states that are reachable from the starting point and realizing that the desired end state is not among them. Pruning adds complexity to this argument since not all reachable states are explored any more. We thus first examine how *blind search*, which does not apply pruning, can create witnesses.

## 6.1 Blind Search

Blind search algorithms such as depth-first search, breadth-first and uniform cost search explore the search space without pruning any states and stop when they have found a goal or all reachable states are explored.

A *progression search* starts at the initial state $I^\Pi$ and expands successors of the considered states at each expansion step. If all (forward) reachable states from $I^\Pi$ have been expanded but no goal is found, the search claims unsolvability. In this case, the set of expanded states serves as an inductive certificate: it contains $I^\Pi$, contains no goal and is inductive.

A *regression search* starts with a partial state denoting all goal states and expands predecessors of the considered partial states. It claims unsolvability if no more partial states can be expanded and the initial state is not subsumed by any expanded partial state. The set of states subsumed by the union of all expanded partial states can be seen as a “backwards inductive certificate” and thus its negation is a (forward) inductive certificate.

Finally, *bidirectional search* has two search frontiers: a forward direction starting from $I^\Pi$ and a backwards direction starting from the goal; expansion steps then advance either the forward or backwards frontier. Unsolvability is claimed if one of the frontiers cannot find any new states. Thus a certificate can be built like in progression if the forward frontier is fully expanded, and as in regression if the backwards frontier is fully expanded.

If we want to generate a proof instead of an inductive certificate, we can simply translate the inductive certificate into a proof as shown in Chapter 5.4.

In what follows we will not distinguish between the depth-first, breadth-first and uniform cost search, since all algorithms must expand all reachable states at some point, and we can add those states to the certificate at this point. We will however distinguish between explicit and symbolic search since states are represented in different ways.

### 6.1.1 Explicit Blind Search

Explicit blind search expands one state at a time. It is most commonly performed in the forward direction, where the set of all expanded states is an inductive certificate. We can build a formula $\varphi$ representing this set by adding each state $s$ to $\varphi$ as it is expanded. For this, we initialize $\varphi$ to $\bot$, and update it the following way when expanding $s$: $\varphi := \varphi \vee (\bigwedge_{v \in s} s \wedge \bigwedge_{v \notin s} \neg v)$. From our considered formalisms, 2CNF and Horn formulas are not suitable since they do not support disjunction, but both MODS and BDD are (MODS works since all formulas consider the same set of variables). In both cases adding the new state can be done in time linear in $|V^\Pi|$: for MODS we can just add the corresponding cube to the formula since all states mention the same set of variables; for BDDs see Theorem A.1 in Appendix A.1.

Explicit blind *regression* search operates on partial states $s_p \subseteq V^\Pi$, which represent all states $s \supseteq s_p$ and can be expressed as formula $\varphi_{s_p} = \bigwedge_{v \in s_p} v$. The inductive certificate in this case consists of all states *not expanded* by the search. This set can be built by a formula $\varphi$ that is initialized to $\top$ and in each expansion step conjoined with the negation of the expanded partial state, i.e. when expanding partial state $s$ we update $\varphi$ the following way: $\varphi := \varphi \wedge \bigvee_{v \in s} \neg v$. 2CNF and MODS are not suitable, since $\bigvee_{v \in s}$ can in general not be represented by 2CNF and requires an exponential representation size in MODS. While BDDs can represent $\varphi$, building it is in general exponential in the number of expanded

states (Edelkamp and Kissmann, 2011). Horn formulas on the other hand can update $\varphi$ in time linear in $|V^{\Pi}|$ and are thus best suited.

### 6.1.2 Symbolic Blind Search

Symbolic search expands an entire set of states in each expansion step. As such, they need to be able to compactly represent state sets. Since most planning systems employing symbolic search like Gamer (Edelkamp and Kissmann, 2009) and SymBA* (Torralba et al., 2014) use BDDs as representation formalism, we only consider inductive BDD certificates.

In contrast to explicit search, symbolic search works principally the same in both progression and regression, since states are represented in an identical way.

Symbolic search stores its closed list in two different ways: For solution reconstruction it keeps a BDD for each distance to the starting set, including all closed states that can be reached from this distance. For duplicate elimination a BDD containing *all* closed states is stored. This BDD is all we need for generating an inductive certificate: In the progression case, it is already an inductive certificate. In the regression case, we simply need to negate the BDD, which takes constant time. Symbolic blind search based on BDDs can thus generate inductive certificates without any overhead.

## 6.2 Heuristic Search

Heuristic search, similarly to blind search, can be performed in various ways. One can choose between progression and regression, explicit and symbolic search, and optimal and suboptimal search. Here we will focus on explicit forward search, where the search starts from the initial state and expands single states based on a combination of its heuristic estimate (an estimate of the distance to the nearest goal state) and possibly other metrics. The exact calculation of the expansion priority does not matter; nor does it matter whether the heuristic estimates are admissible or not. The only property we require of the heuristic is *safety*: it can only declare a state $s$ as *dead-end* if no goal can be reached from $s$.

Heuristic search changes the detection of unsolvabilty insofar that not the entire reachable search space must be explored anymore. If a heuristic detects a state as *dead-end*, i.e. it detects that no goal can be reached via this state, the state can be pruned from the search without affecting correctness. Since the dead-end state cannot reach any goal, we know that an inductive set containing the dead-end but no goal state must exist. While the next chapter focuses on how to describe such a set for a multitude of heuristics, we will currently assume that these sets are given, i.e. we assume that for every state $d$ pruned during search we have an inductive set $S_d$ containing $d$ but no goal states.[1]

[1]For proofs in the proof system it is already sufficient to have some form of proof why $d$ is dead (which can but does not necessarily need to argue with inductive sets), but since we consider both types of witnesses here, we assume a stricter requirement.

The key of proving unsolvability now lies in combining these inductive sets with the set of expanded states. Intuitively, if dead-ends cannot reach a goal and the expanded states can only reach themselves and dead-ends it is clear that the task cannot be solved. In the next two sections we will analyze separately how inductive certificates and proofs in the proof system can be built from this.

### 6.2.1 Inductive Certificates

We can form an inductive certificate with the union of sets $S_d$ for all dead-ends $d$ and the set of expanded states, since it is an overapproximation of the states reachable from $I^\Pi$, and all states in this union that are not reachable from $I^\Pi$ must be contained in some inductive set $S_d$. However, we can in general not generate a single $\mathbf{R}$-formula representing this union, because we need equally many disjunctions as dead-ends occurred. Since often several thousand dead-ends are detected in a heuristic search, we cannot rely on bounded disjunction anymore but must require $\vee\mathbf{C}$, which none of our considered formalisms supports.

Instead we suggest to build a disjunctive certificate consisting of the sets $S_d$ for dead-ends $d$ and for each expanded state $e$ a separate set. This family of sets forms a *1-disjunctive* certificate:

**Theorem 6.1.** *Let $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$ be a STRIPS planning task, let $S_\infty \subseteq S^\Pi$ be a set of dead-ends, and let $\mathcal{F}_\infty$ be a family of inductive sets such that $\bigcup_{S\in\mathcal{F}_\infty} S \cap S^\Pi_G = \emptyset$ and $S_\infty \subseteq \bigcup_{S\in\mathcal{F}_\infty} S$ (i.e. each state in $S_\infty$ is included in some inductive set in $\mathcal{F}_\infty$).*

*Moreover, let $S_{\text{exp}} \subseteq S^\Pi$ be a set of states such that $I^\Pi \in S_{\text{exp}} \cup S_\infty$, $S_{\text{exp}}$ contains no goal state, and $S_{\text{exp}}[A^\Pi] \subseteq S_{\text{exp}} \cup S_\infty$. Then $\mathcal{F} = \{\{s\} \mid s \in S_{\text{exp}}\} \cup \mathcal{F}_\infty$ is a 1-disjunctive certificate for $\Pi$.*

**Proof:** We check the three requirements for 1-disjunctive certificates in Definition 4.5. Property 1 (inclusion of the initial state) and 2 (non-inclusion of goal states) are trivial. It remains to check property 3 (disjunctive 1-inductivity) for all component certificates $S \in \mathcal{F}$ and all actions $a \in A^\Pi$.

*Case 1:* $S = \{s\}$ for some $s \in S_{\text{exp}}$. Then $S[a]$ is either empty (if $a$ is inapplicable in $s$) and the condition holds trivially, or $S[a] = \{s[a]\}$, and we can set $\mathcal{F}'$ in Definition 4.5 to include some set in $\mathcal{F}$ that includes $s[a]$. Such a set must exist because $S_{\text{exp}}[A^\Pi] \subseteq S_{\text{exp}} \cup S_\infty$ and $S_\infty \subseteq \bigcup_{S\in\mathcal{F}_\infty} S$.

*Case 2:* $S \in \mathcal{F}_\infty$. Then we can set $\mathcal{F}' = \{S\}$ in Definition 4.5 because $S$ is inductive. □

Note that while building one set for all expanded states would seem more intuitive it would violate 1-disjunctiveness, since applying an action to two different expanded states might lead to a dead-end in one case and an expanded state in the other case; or to two different dead-ends covered by two different inductive sets. Figure 6.1 illustrates this: If we had a single set $S = \{I^\Pi, s_1, s_2, s_3\}$ containing all expanded states, then for example $S[a_2]$ has elements in $S$, $S_{d_1}$ and $S_{d_2}$ and thus the certificate is no longer 1-disjunctive.

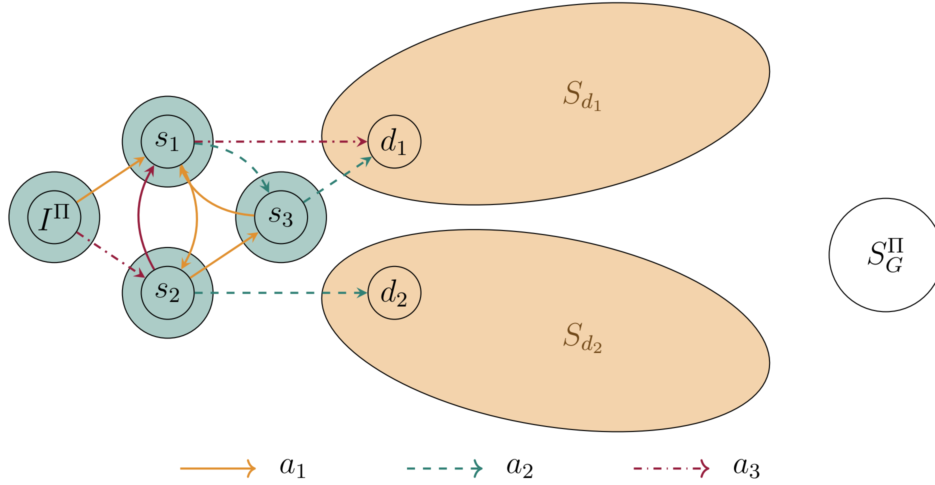



Figure 6.1: Example of a search space where a heuristic recognized $d_1$ and $d_2$ as dead-ends. The red sets denote inductive sets $S_d$ obtained by the heuristic for each dead-end, and blue sets are singleton sets, each including exactly one expanded state.

A set $S = \{s\}$ is represented by a formula $\bigwedge_{v \in s} v \wedge \bigwedge_{v' \notin s} \neg v'$ and requires only **CL**, thus all considered representation formalisms are suitable. However, since those sets form only part of the certificate there are still some limitations:

- Each set for a dead-end must be represented explicitly by one single formula.
- The representation formalism must be the same for *all* sets. For BDDs, this also means all sets must have the same variable ordering.

The second point also implies that building certificates for heuristic search with several heuristics is in general not covered by this approach, even if all heuristics are supported in isolation but require different formalisms.

**Theorem 6.2.** *Given an unsolvable STRIPS planning task $\Pi$, an explicit heuristic forward search with heuristic $h$ can generate a 1-disjunctive $\mathbf{R}$-certificate iff $\mathbf{R}$ efficiently supports **CL** and $h$ can generate an inductive set represented by $\mathbf{R}$-formula $\varphi_d$ for each detected dead-end $d$ such that $d \in$ states$(\varphi_d)$ and $S_G^\Pi \cap$ states$(\varphi_d) = \emptyset$.*

All four considered formalisms are in principle suitable, since they efficiently support **CL** and can efficiently verify 1-disjunctive certificates. In the case of MODS we do not even need the restriction that all formulas are over the same set of variables, since for 1-disjunctive certificates the disjunction on the right side contains only one element.

### 6.2.2 Proof System

As we have shown in Theorem 5.10, we could simply translate the 1-disjunctive certificate from above into our proof system. Such a proof would however be considerably bigger than the certificate, and more importantly we would not make use of the richer expressiveness of the proof system and impose the same restrictions on which algorithms could produce proofs at all.

The main feature of the proof system is to be able to combine knowledge gained from different sources. Following this design, the key idea of generating proofs from heuristic search is to first show for each individual dead-end why it is dead, and then combine this knowledge by showing that the set of expanded states only leads to itself or to dead states, meaning we cannot reach any goal from the initial state.

**Theorem 6.3.** *Given an unsolvable STRIPS planning task $\Pi$, an explicit heuristic forward search can generate a proof of unsolvability iff each heuristic can prove for each detected dead-end $d$ that some set expression $S_d$ containing $d$ is dead, where all set variables in $S_d$ are represented by some $\mathbf{R}$-formalism and no progression or regression occurs. To generate the proof, we additionally need a formalism $\mathbf{R}'$ that efficiently supports **CL** and is able to build a formula by iteratively adding models.*

**Proof:** In the first part of the proof we prove for each dead-end $d$ in the set of all dead-ends $S_\infty$ that the set $\{d\}$ represented in $\mathbf{R}'$ is dead. We are already given formulas $\varphi_\mathbf{R}$ representing a set $S_d$ that has been proven dead and know that $d \in S_d$ holds. First, we represent $\{d\}$ by building an $\mathbf{R}'$-formula $\psi_{\mathbf{R}'} = \bigwedge_{v \in d} v \wedge \bigwedge_{v \notin d} \neg v$, using **CL**. If $S_d$ is a set literal, we can directly show that $\{d\} \sqsubseteq S_d$ holds with basic statement **B4**. Otherwise $S_d$ must be union or intersection of two sets and we recurse over these two sets. If $S_d = (S \cup S')$, then $\{d\} \sqsubseteq S$ or $\{d\} \sqsubseteq S'$ must hold since $\{d\}$ contains only a single element. After recursively showing either statement, we can use **UR** or **UL** together with **ST** to show $\{d\} \sqsubseteq (S \cup S')$. If $S_d = (S \cap S')$, then both $\{d\} \sqsubseteq S$ and $\{d\} \sqsubseteq S'$ must hold. After recursively showing both statements we use **SI** to get $\{d\} \sqsubseteq (S \cap S')$. Once we obtained $\{d\} \sqsubseteq S_d$, we can use the existing knowledge that $S_d$ is dead by applying rule **SD** to get (1) $\{d\}$ is dead.

We now have a set $\{d\}$ represented by an $\mathbf{R}'$-formula for each $d \in S_\infty$, and we know all such sets are dead. We combine this knowledge by applying rule **UD** $|S_\infty| - 1$ times in order to get the statement (2) $\bigcup_{d \in S_\infty} \{d\}$ is dead.

For the second part of the proof, we represent the set of expanded states $S_{\text{exp}}$ as an $\mathbf{R}'$-formula, which is built incrementally each time a state is expanded. We know from the search that expanded states only lead to expanded states or dead-ends and state this with **B2** (3) $S_{\text{exp}}[A^\Pi] \sqsubseteq S_{\text{exp}} \cup \bigcup_{d \in S_\infty} \{d\}$. We also know that $S_{\text{exp}}$ cannot contain any goal states since otherwise the task would not be unsolvable, which leads to statement **B1** (4) $S_{\text{exp}} \cap S_G^\Pi \sqsubseteq \emptyset$. From (4) together with rule **ED** ($\emptyset$ dead) follows (5) $S_{\text{exp}} \cap S_G^\Pi$ dead. Now we can conclude from (3), (2) and (5) that (6) $S_{\text{exp}}$ is dead with **PG**. Since $I^\Pi$ must have been expanded we can state (7) $\{I^\Pi\} \sqsubseteq S_{\text{exp}}$ with **B1**, and together with (6) follows that (8) $\{I^\Pi\}$ is dead according to **SD**. Applying **CI** to (8) concludes the proof.

For the special case where the initial state is already detected to be a dead-end, we can directly conclude the proof with **CI** applied to (1). In this case, we do not need need $\mathbf{R}'$ and $\{I^\Pi\} \sqsubseteq S_d$ is covered by basic statement **B1** and **CL**. □

For our four considered formalisms, this concretely means that only BDDs or MODS can take on the role of $\mathbf{R}'$ because only they can incrementally add models to a formula in an efficient manner (see Section 6.1.1).

For verifying the proof, basic statement **B2** in (3) can become problematic, since the size of the union is linear in the amount of dead-ends. We observe however that the union represents a disjunction over models with the same variables, and thus both MODS and BDDs can efficiently build this union. Alternatively we can split up the role of $\mathbf{R}'$ by building a single set $D_{\mathbf{R}'}$ of *all* dead-ends in $\mathbf{R}'$ and show with **B1** that it is a subset of the union of all $\{d\}$, and then build a set $D_{\mathbf{R}''}$ containing the same states in a different formalism $\mathbf{R}''$ and show with **B4** that $D_{\mathbf{R}''} \sqsubseteq D_{\mathbf{R}'}$ holds. The remainder of the proof can then be shown using $\mathbf{R}''$, which does not need to be able to build a union of models.

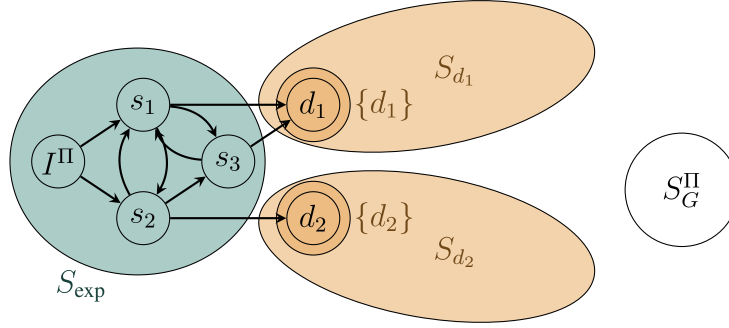
While we cannot discuss all cases how a heuristic might show that $S_d$ is dead, we analyze the case where $S_d$ is represented by a single $\mathbf{R}$-formula $\varphi_{\mathbf{R}}$ and must be inductive and contain no goal states. Showing that $S_d$ is dead can then be done the following way:

We first establish $S_d \cap S_G^\Pi \sqsubseteq \emptyset$ with basic statement **B1**, and by applying **SD** together with **ED** ($\emptyset$ dead) we have (1) $S_d \cap S_G^\Pi$ dead. Inductivity means that we can state $S_d[A^\Pi] \sqsubseteq S_d$ with **B2**, which we can extend to (2) $S_d[A^\Pi] \sqsubseteq (S_d \cup \emptyset)$ with the help of **UR** ($S_d \sqsubseteq (S_d \cup \emptyset)$) and **ST**. We then can apply rule **PG** to (1), **ED** and (2) to get $S_d$ dead.

For basic statement **B2** $S_d[A^\Pi] \sqsubseteq S_d$ we need $\wedge$**BC**, **CL**, **RN**$_\prec$ and **SE**, which also covers **B1** $S_d \cap S_G^\Pi \sqsubseteq \emptyset$.

**Corollary 6.1.** *If a heuristic can produce an* $\mathbf{R}$*-formula* $\varphi_R$ *representing an inductive set* $S_d$ *containing no goal state, we can generate a proof showing* $S_d$ *is dead that can be efficiently verified iff* $\mathbf{R}$ *efficiently supports* $\wedge$***BC**, **CL**, **RN***$_\prec$ *and **SE**.*

Figure 6.2 shows an example of a full proof where each $S_d$ is inductive, contains no goal state and is represented by a single $\mathbf{R}$-formula. We can build the proof efficiently by creating sets $S_{d_i}$ and $\{d_i\}$ and showing they are dead each time a dead-end is detected, and by iteratively building $S_{\text{exp}}$ every time a state is expanded.




| # | judgment | rule | premises |
|---|---|---|---|
| (1) | $\emptyset$ dead | **ED** | |
| (2) | $S_{d_1}[A^\Pi] \sqsubseteq S_{d_1}$ | **B2** | |
| (3) | $S_{d_1} \sqsubseteq (S_{d_1} \cup \emptyset)$ | **UR** | |
| (4) | $S_{d_1}[A^\Pi] \sqsubseteq (S_{d_1} \cup \emptyset)$ | **ST** | (2),(3) |
| (5) | $S_{d_1} \cap S_G^\Pi \sqsubseteq \emptyset$ | **B1** | |
| (6) | $S_{d_1} \cap S_G^\Pi$ dead | **SD** | (1),(5) |
| (7) | $S_{d_1}$ dead | **PG** | (4),(1),(6) |
| (8) | $\{d_1\} \sqsubseteq S_{d_1}$ | **B4** | |
| (9) | $\{d_1\}$ dead | **SD** | (7),(8) |
| (10) | $S_{d_2}[A^\Pi] \sqsubseteq S_{d_2}$ | **B2** | |
| (11) | $S_{d_2} \sqsubseteq (S_{d_2} \cup \emptyset)$ | **UR** | |
| (12) | $S_{d_2}[A^\Pi] \sqsubseteq (S_{d_2} \cup \emptyset)$ | **ST** | (10),(11) |
| (13) | $S_{d_2} \cap S_G^\Pi \sqsubseteq \emptyset$ | **B1** | |
| (14) | $S_{d_2} \cap S_G^\Pi$ dead | **SD** | (1),(13) |
| (15) | $S_{d_2}$ dead | **PG** | (12),(1),(14) |
| (16) | $\{d_2\} \sqsubseteq S_{d_2}$ | **B4** | |
| (17) | $\{d_2\}$ dead | **SD** | (15),(16) |
| (18) | $(\{d_1\} \cup \{d_2\})$ dead | **UD** | (9),(17) |
| (19) | $S_{\text{exp}}[A^\Pi] \sqsubseteq (S_{\text{exp}} \cup (\{d_1\} \cup \{d_2\}))$ | **B2** | |
| (20) | $S_{\text{exp}} \cap S_G^\Pi \sqsubseteq \emptyset$ | **B1** | |
| (21) | $S_{\text{exp}} \cap S_G^\Pi$ dead | **SD** | (1),(20) |
| (22) | $S_{\text{exp}}$ dead | **PG** | (19),(18),(21) |
| (23) | $\{I^\Pi\} \sqsubseteq S_{\text{exp}}$ | **B1** | |
| (24) | $\{I^\Pi\}$ dead | **SD** | (22),(23) |
| (25) | task unsolvable | **CI** | (24) |

Set variables: $S_{d_1} : \mathbf{R}_1 \quad S_{d_2} : \mathbf{R}_2 \quad \{d_1\} : \mathbf{R}' \quad \{d_2\} : \mathbf{R}' \quad S_{\text{exp}} : \mathbf{R}'$

Figure 6.2: An example of how a proof for heuristic search can be built. The heuristics detected $d_1$ and $d_2$ as dead-end, and provided two sets $S_{d_1}$ and $S_{d_2}$ that include the corresponding dead-end, do not contain any goal state and are inductive.

# 7 Heuristics

For the purpose of detecting unsolvability, heuristics only serve a pruning purpose: when a heuristic returns an infinite estimate for a state $s$, it concluded that no goal state can be reached from $s$ and thus $s$ does not need to be expanded. In the previous chapter we discussed how heuristic search can emit a certificate or proof of unsolvability, based on the assumption that if a heuristic reports a dead-end it can produce an inductive set containing the dead-end but no goal states (for inductive certificates) or can prove some set $S_d$ containing the dead-end dead (for proofs). In this chapter we now show for several heuristics how such a set can be generated.

While the proof approach does not require $S_d$ to be inductive and contain no goal (it only needs to show in some way that $S_d$ is dead), all heuristics we discuss in this chapter can provide a set with these properties. We thus restrict our discussion to such sets, which we call *dead-end validations*:

**Definition 7.1** (dead-end validation)**.** *Given a heuristic $h$ and a state $s$ that $h$ claims to be a dead-end, a* dead-end validation *is a set $S$ of states with the following properties:*

- $s \in S$
- $S \cap S_G^\Pi = \emptyset$
- *$S$ is inductive*

If we consider the set of all states that a given heuristic detects as dead, certain properties of a heuristic directly imply properties of the dead-end validation. Obviously, the first property is given since we consider the set of all dead-ends. For the second and third property, we look at the following heuristic properties:

- safety: A heuristic $h$ is *safe* if it only declares a state $s$ as dead-end if no goal can be reached from $s$.
- consistency: A heuristic $h$ is *consistent* if $h(s) \leq h(s[a]) + \mathit{cost}(a)$ for any state $s$ and applicable action $a$, where *cost* : $A \rightarrow \mathbb{R}_0^+$ assigns each action a finite nonnegative cost.

The safety property implies that no goal is in the set of all dead-ends, and the consistency property implies that this set is inductive:

**Theorem 7.1.** *Given a heuristic $h$, let $H_{inf}$ be the set of states $s \in S^\Pi$ with $h(s) = \infty$.*

*(a) If $h$ is safe, $H_{inf}$ contains no goal state.*

*(b) If $h$ is consistent, $H_{inf}$ is inductive.*

**Proof:** (a) If $h$ is safe, then it can only report states as dead-end if no goal can be reached from them. Thus if $s \in H_{\text{inf}}$, then no goal state can be reached from $s$, which also means that $s$ cannot be a goal state. (b) From the definition of consistency follows that if $h(s) = \infty$, then $h(s') = \infty$ for all $s'$ reachable from $s$, i.e. all states reachable from any $s \in H_{\text{inf}}$ must also be in $H_{\text{inf}}$. Thus $H_{\text{inf}}$ is inductive. □

Theorem 7.1 shows that for consistent and safe heuristics we can define one singular set that contains all dead-ends and serves as a dead-end validation for all dead-ends. However, such a set can not always be represented compactly. We thus also explore options where we build different sets for each dead-end.

## 7.1 Delete Relaxation

The family of delete relaxation heuristics like $h^+$, $h^{\text{max}}$, $h^{\text{add}}$, $h^{\text{FF}}$ and $h^{\text{LM-Cut}}$ (e.g. Bonet and Geffner, 2001; Hoffmann and Nebel, 2001; Helmert and Domshlak, 2009) operates on an altered planning task where the delete effects of all actions are removed:

**Definition 7.2** (delete relaxation)**.** *Let $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$ be a STRIPS planning task. The delete relaxation $\Pi^R = \langle V^\Pi, A^R, I^\Pi, G^\Pi \rangle$ of $\Pi$ contains an action $a^R \in A^R$ for each $a \in A^\Pi$ such that $pre(a^R) = pre(a)$, $add(a^R) = add(a)$ and $del(a^R) = \emptyset$.*

We say that a variable $v$ is *unreachable in $\Pi^R$ from state $s$* if it is impossible to reach a state from $s$ that contains $v$, and call the set of all such variables $U_s^R$. Since in delete free tasks an action applicable to a state $s$ is applicable to all successors of $s$, we can incrementally compute $U_s^R$ by applying all applicable actions until we reach a fix point $s^*$. The set of all variables not contained in this fix point is $U_s^R$.

In the original task $\Pi$ this set of variables can be used to define inductive sets:

**Theorem 7.2.** *Given $U_s^R$ for some state $s$, the set $S_s^R = \{s' \subseteq V^\Pi | s' \cap U_s^R = \emptyset\}$ is inductive in $\Pi$*

**Proof:** We know that in $\Pi^R$ the state $s^* = V^\Pi \setminus U_s^R$ is a fix point, i.e. $s^*[a^R] = s^*$ if $a^R$ is applicable. In $\Pi$, the corresponding actions $a$ are applicable to $s^*$ since their preconditions are identical and they must lead to a state $s' \subseteq s^*$ (or $s' \in S_s^R$) since the add effects are identical but $a$ might contain delete effects, meaning $\{s^*\}[A^\Pi] \subseteq S_s^R$. Furthermore, for any state $s'' \subseteq s^*$ we have that $s''[a] \subseteq s^*[a]$ (and thus $s''[a] \in S_s^R$) if $a$ is applicable in $s''$ due to the relaxation lemma [1]. From this we conclude that $S_s^R[A^\Pi] \subseteq S_s^R$. □

[1] The relaxation lemma (for STRIPS) states that given two states $s \subseteq s'$ and action sequence $\pi$ applicable to $s$, $\pi$ is also applicable to $s'$ and $s[\pi] \subseteq s'[\pi]$.

Any delete relaxation heuristic detects a state $s$ only as dead-end if $U_s^R$ contains at least one goal variable. In this case $S_s^R$ contains no goal states, meaning it is a dead-end validation for $s$.

$S_s^R$ can be represented by $\bigwedge_{v \in U_s^R} \neg v$, requiring only **CL**, which all considered formalisms support efficiently. Aside from generating the set with **CL**, no additional overhead is incurred since delete relaxation heuristics perform a relaxed reachability analysis as part of their heuristic computation.

**Corollary 7.1.** *Given a state $s$ that is detected as dead-end by a delete relaxation heuristic, a dead-end validation for $s$ can be represented by a single $\mathbf{R}$-formula $\varphi_{\mathbf{R}}$ iff $\mathbf{R}$ efficiently supports* ***CL****.*

This means that all four considered formalisms are suitable for delete-relaxation heuristics. Since the dead-end validation is represented as a single formula $\varphi_{\mathbf{R}}$ it can be used for both proofs (with Corollary 6.1) as well as 1-disjunctive $\mathbf{R}$-certificates.

## 7.2 Critical Paths

The family of $h^m$ (Haslum and Geffner, 2000) heuristics are a generalization of the $h^{\max}$ heuristic and can intuitively be understood to relax the task by considering the reachability of variable tuples with size up to $m$. Similar to delete relaxation heuristics, $h^m$ computes a set of reachable variable tuples as a side effect, and we can show that the set of all states containing only tuples from this set is inductive.

In order to prove this claim we utilize the $\Pi^m$ compilation introduced by Haslum (2009), which is defined such that $h^{\max}(\Pi^m) = h^m(\Pi)$, i.e. instead of directly computing the $h^m$ heuristic we can compute the $h^{\max}$ heuristic on $\Pi^m$. It explicitly represents conjunctions $c \subseteq V^\Pi$ of up to $m$ variables by new variables $\pi_c$. We use the shorthand $X^m$ to denote the set of such new variables for all conjunctions of up to $m$ variables that are implied by a set of variables $X \subseteq V^\Pi$: $X^m = \{\pi_c \mid c \subseteq X \text{ and } |c| \leq m\}$. With this shorthand, we can define the $\Pi^m$ compilation as follows:

**Definition 7.3** ($\Pi^m$ compilation)**.** *For a STRIPS planning task $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$ and a parameter $m \in \mathbb{N}^+$, $\Pi^m$ is the planning task $(V^m, A^m, I^m, G^m)$, where $A^m$ contains an action $a^f$ for each pair $a \in A^\Pi$ and $f \subseteq V^\Pi$ such that $|f| < m$ and $f$ is disjoint from $del(a) \cup add(a)$. Action $a^f$ is given by*

$$pre(a^f) = (pre(a) \cup f)^m$$
$$add(a^f) = (add(a) \cup f)^m$$
$$del(a^f) = \emptyset$$

The original definition in Haslum (2009) does not include variables $\pi_c$ in $add(a^f)$ where $c \cap add(a) = \emptyset$. For these variables, it holds that $c \subseteq f$, so $\pi_c$ is included in $pre(a^f)$. Our additional add effects do therefore not change the semantics of the actions.

We say that a variable $\pi_c$ is unreachable in $\Pi^m$ from state $s$ if it is impossible to reach a state from $s$ where $\pi_c$ is true, and call the set of all such variables $U_s^m$. For a given $\pi_c$, the set $S_{\bar{c}} = \{s \subseteq V^\Pi \mid c \not\subseteq s\}$ denotes the set of all states in $\Pi$ that do not contain all variables in $c$. The intersection of these sets for all $\pi_c$ in $U_s^m$ is inductive:

**Theorem 7.3.** *Given a task $\Pi$ and a state $s$, the intersection of the family of sets $\mathcal{F} = \{S_{\bar{c}} \mid \pi_c \in U_s^m\}$ is inductive.*

**Proof:** We will show that for each action $a \in A^\Pi$ and $S_{\bar{c}} \in \mathcal{F}$ there is a $c'$ such that $S_{\bar{c}'}[a] \subseteq S_{\bar{c}}$. As we have seen in the proof to Theorem 4.7, this implies that $\bigcap_{S_{\bar{c}} \in \mathcal{F}} S_{\bar{c}}$ is inductive.

Consider an action $a$ and $S_{\bar{c}} \in \mathcal{F}$. If $\mathit{add}(a) \cap c = \emptyset$ or $\mathit{del}(a) \cap c \neq \emptyset$ then $a$ cannot make all variables in $c$ true and $S_{\bar{c}}[a] \subseteq S_{\bar{c}}$. Otherwise, let $f := c \setminus \mathit{add}(a)$ be the set of variables that needs to be true in a state $s'$ such that $s'[a] \notin S_{\bar{c}}$. Consider action $a^f$ of the task $\Pi^m$. It holds that $\pi_c \in \mathit{add}(a^f)$ but $\pi_c \in U_s^m$, so $\mathit{pre}(a^f)$ cannot be relaxed reachable from $s^m$ in $\Pi^m$, i.e. there is a $\pi_{c'} \in U_s^m \cap \mathit{pre}(a^f)$. From $\pi_{c'} \in \mathit{pre}(a^f) = (\mathit{pre}(a) \cup f)^m$, we know that $c' \subseteq \mathit{pre}(a) \cup f$, which means that for any state in $S_{\bar{c}'}$ we have that either $a$ is not applicable or not all variables in $f$ are true (and thus in the successor not all variables of $c$ are true). From this we conclude $S_{\bar{c}'}[a] \subseteq S_{\bar{c}}$. □

$h^m$ detects a state as dead-end iff some $\pi_c$ with $c \subseteq G$ is in $U_s^m$. In this case, the intersection of the sets in $\mathcal{F}$ does not contain any goal state and is thus a dead-end validation.

This dead-end validation can be built with $\mathbf{R}$-formula $\varphi = \bigwedge_{\pi_c \in U_s^m} \bigvee_{v \in c} \neg v$, requiring $\wedge\mathbf{C}$ and the ability to represent a clause containing only negated variables. This means only Horn formulas are suitable, or 2CNF if $m \leq 2$. Alternatively we can define it as the intersection of sets represented by $\mathbf{R}$-formulas $\varphi_c = \bigvee_{v \in c} \neg v$ for all $\pi_c \in U_s^m$. For this, BDDs and MODS are suitable as well. In both cases we can build the formula(s) in $O(|m| \cdot |U_s^m|)$.

**Corollary 7.2.** *Given a state $s$ that is detected as dead-end by an $h^m$ heuristic, a dead-end validation for $s$ can be represented by a single formula $\bigwedge \bigvee \neg v$, i.e. a CNF formula where all literals are negated variables.*

Looking at our considered formalisms, this means that 1-disjunctive certificates can be built for heuristic search with an $h^m$ heuristic, but only represented by Horn formulas, or if $m \leq 2$ by 2CNF formulas. For proofs, we can show that the singular formula representation is dead with Corollary 6.1. We can however also avoid building the singular formula. As we have seen in the proof for Theorem 7.3, for each $S_{\bar{c}}$ (represented by $\varphi_c$) and action $a \in A^\Pi$, we can find a $S_{\bar{c}'}$ such that $S_{\bar{c}'}[a] \subseteq S_{\bar{c}}$, which is reminiscent of 1-conjunctive certificates. Indeed, if the initial state is detected as a dead-end by $h^m$, we could build a 1-conjunctive certificate with these formulas. With the proof system, we can build a proof for *any* state $s$ detected as dead-end by $h^m$ showing that $s$ is dead analogously to how we translate $r$-conjunctive certificates into proofs.

**Corollary 7.3.** *Given a state $s$ that is detected as dead-end by an $h^m$ heuristic, we can build a proof that $s$ is dead in any $\mathrm{R}$-formalism that can represent a clause containing only negated variables. This proof can be verified efficiently if $\mathrm{R}$ efficiently supports **CE**, **SE**, $\wedge$**BC**, **CL** and **RN**$_{\prec}$.*

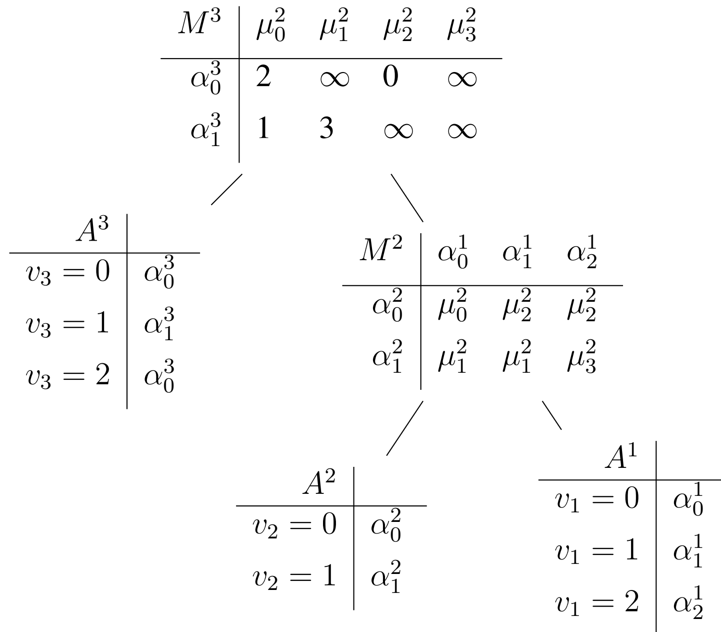
## 7.3 Merge and Shrink

Pattern databases (PDBs) (Edelkamp, 2001) and their generalization the M&S heuristic (Helmert, Haslum, and Hoffmann, 2007) are defined in a SAS$^+$ setting. Their general idea is to build an abstraction of the search space and use the goal distance in the abstraction as heuristic estimate. Pattern databases abstract the planning problem to a set of variables (called the "pattern"). The M&S heuristic first builds abstractions for all variables (called atomic abstractions), and then iteratively *merges* two abstractions, potentially *shrinking* the result, until only one abstraction remains. The *merge strategy* (represented by a tree) dictates in which order abstractions are merged. If the tree degenerates to a list (meaning one abstraction in the merge step is always an atomic abstraction), we speak of a *linear merge strategy*. PDBs can also be understood as an M&S heuristic using a linear merge strategy and abstracting all variables not in the pattern to only one abstract state.

M&S heuristics internally use a cascading tables representation, where each node in the merge tree is associated with a table. Tables for leaf nodes map each value of the associated SAS$^+$ variable to an abstract state in the atomic abstraction. Tables for merge nodes map pairs of abstract states from its two child nodes to new abstract states, except for the top-level node which maps to heuristic values instead. The heuristic value for a state $s$ can then be looked up by first looking up the abstract states for each variable and then following the merge tree. As an example consider Figure 7.1: Given a state $s = \{v_1 = 0, v_2 = 1, v_3 = 2\}$, we see that $v_1 = 0$ maps to $\alpha_0^1$ in leaf table $A^1$, $v_2 = 1$ maps to $\alpha_1^2$ in $A^2$, and $v_3 = 2$ to $\alpha_0^3$ in $A^3$. In merge table $M^2$ we see that $\langle \alpha_1^2, \alpha_0^1 \rangle$ maps to $\mu_1^2$, and in $M^3$ (the final table) $\langle \alpha_0^3, \mu_1^2 \rangle$ maps to $\infty$, thus $h(s) = \infty$.

In order to reduce the number of abstract states, the shrinking step maps all states of an abstraction that either (i) cannot reach any abstract goal state or (ii) cannot be reached by the abstract initial state to one abstract "dead state", which will result in the corresponding concrete states having a heuristic value of $\infty$ and effectively being pruned from the search. Since condition (ii) results in states from which a goal can be reached to have infinite heuristic value, M&S heuristics are in general neither safe nor consistent. If we however do not prune states that only satisfy condition (ii) (i.e. states from which a goal can be reached but that cannot be reached from the initial state), the resulting M&S (or PDB) heuristic is safe and consistent. In what follows we only consider this variant of M&S heuristics.

Since the considered M&S heuristics are consistent and safe, the set of all states with infinite heuristic estimate is inductive and does not contain any goal. While for general heuristics it is hard to represent this set, Helmert et al. (2014) have shown that for *linear*



| $M^3$ | $\mu^2_0$ | $\mu^2_1$ | $\mu^2_2$ | $\mu^2_3$ |
|---|---|---|---|---|
| $\alpha^3_0$ | 2 | $\infty$ | 0 | $\infty$ |
| $\alpha^3_1$ | 1 | 3 | $\infty$ | $\infty$ |

| $A^3$ | |
|---|---|
| $v_3 = 0$ | $\alpha^3_0$ |
| $v_3 = 1$ | $\alpha^3_1$ |
| $v_3 = 2$ | $\alpha^3_0$ |

| $M^2$ | $\alpha^1_0$ | $\alpha^1_1$ | $\alpha^1_2$ |
|---|---|---|---|
| $\alpha^2_0$ | $\mu^2_0$ | $\mu^2_2$ | $\mu^2_2$ |
| $\alpha^2_1$ | $\mu^2_1$ | $\mu^2_1$ | $\mu^2_3$ |

| $A^2$ | |
|---|---|
| $v_2 = 0$ | $\alpha^2_0$ |
| $v_2 = 1$ | $\alpha^2_1$ |

| $A^1$ | |
|---|---|
| $v_1 = 0$ | $\alpha^1_0$ |
| $v_1 = 1$ | $\alpha^1_1$ |
| $v_1 = 2$ | $\alpha^1_2$ |

Figure 7.1: An example showing the cascading tables representation of a M&S heuristic with linear merge strategy.

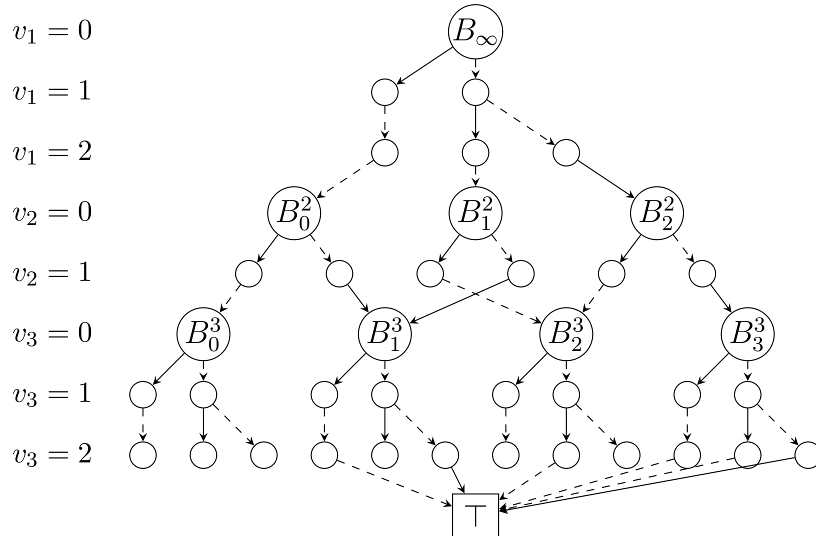



Figure 7.2: The (partially reduced) BDD representing all dead states of the M&S heuristic from Figure 7.1. Leaf node $\bot$ and all edges leading to it are omitted for clarity.

*merge strategies* we can efficiently build an ADD[2] (Bahar et al., 1993) from the cascading tables. From this ADD we can efficiently extract a BDD representing all states mapping to $\infty$. We need to consider however that the M&S heuristic is defined on SAS$^+$ and our translation to STRIPS introduces new states, thus Theorem 7.1 can not be applied directly. But due to the construction of the STRIPS task it is easy to see that for any state for which the M&S heuristic is defined (i.e. all states not violating the mutexes inherent in the SAS$^+$ task) the successor states all also do not violate these mutexes. Thus the set of states for which the M&S heuristic is defined and evaluates to $\infty$ is indeed inductive even for the translated STRIPS task.

When extracting the BDD with all dead states we largely follow the construction from Helmert et al. (2014) except for two changes: (1) we build the BDD directly from the cascading tables, and (2) our binary representation of SAS$^+$ variable $v_i$ consists of the $|dom(v_i)|$ propositional variables from our STRIPS translation (instead of $|dom(v_i)| - 1$ variables). Algorithm 1 shows how we construct the BDD and an example can be seen in Figure 7.2. The construction centers around intermediate BDDs $B^i_j$, which relate to the abstract states $j$ of the right child of $M^i$. It represents how, given $j$, remaining variables $v_{k'}$ with $k' \geq i$ must be assigned such that the resulting state is dead. The variable ordering for the BDDs is taken from the (linear) merge strategy. We denote variables in such a way that $h^{M\&S}$ first merges $v_1$ and $v_2$, then merges the result with $v_3$ and so on. The variable ordering for our BDD then is $v_{1,0} \prec \cdots \prec v_{1,|dom(v_1)|-1} \prec v_{2,1} \prec \cdots \prec v_{n,|dom(v_n)|-1}$ (assuming $dom(v_i) = \{0, |dom(v_i)| - 1\}$ for all variables $v_i \in V_{\text{SAS}^+}$).

We first replace all finite heuristic values in the last merge table $M^n$ with 0 and can thus interpret $M^n$ to map to the two abstract states 0 and $\infty$. Since we want to build a BDD representing all dead states, we assign $B^{n+1}_0 = \bot$ and $B^{n+1}_\infty = \top$. We then iterate backwards over the merge tables $M^i$ and build a BDD $B^i_j$ for each column of $M^i$ in the following way: We build a BDD for each value $x$ in $dom(v_i)$ (i.e. the domain of the variable in the left leaf child) and combine it with $B^{i+1}_{j'}$, i.e. the BDD representing the abstract state $j'$ mapped to by $j$ and the abstract state for $x$ in $M^i$. The union over these BDDs forms $B^i_j$. After we processed the final merge table $M^2$, BDDs $B^2_j$ correspond to abstract states of the atomic abstraction to $v_1$. We build the final BDD by again combining each value of $v_1$ with the appropriate $B^2_j$ BDD and building the overall union.

Each BDD operation in lines 9–10 and 17–18 take time linear in $dom(v_i)$:

- $B_{v_i=k}$: Building a partial variable assignment to $k$ variables is linear in $k$, and $B_{v_i=k}$ is a partial assignment to $dom(v_i)$ variables (independent of variable ordering).
- With Theorem A.3 from Appendix A.1 we know that since all variables occurring in $B_{v_i=k}$ are ordered before any in $B^{i+1}_{\text{entry}}$, the conjunction $B_{v_i=k} \wedge B^{i+1}_{\text{entry}}$, is done in time linear in $\|B_{v_i=k}\| = dom(v_i)$.

[2]Algebraic Decision Diagrams (ADDs) have a finite amount of leaf nodes representing integers or $\infty$ as opposed to BDD with leaves $\top$ and $\bot$.

**Algorithm 1:** Build a BDD representing all dead states of a given M&S heuristic.

**Data:** Cascading tables $A^1, \dots, A^n$ and $M^2, \dots, M^n$. $M^i$ has left child $A^i$ and right child $M^{i-1}$ for $i > 2$; $M^2$ has left child $A^2$ and right child $A^1$. All $A^i$ are leaves.

**Result:** BDD $B_\infty$

```
1  replace finite values in M^n with 0;
2  B_0^{n+1} ← ⊥;
3  B_∞^{n+1} ← ⊤;
4  for i = n...2 do
5      for j ∈ M^i.column do
6          B_j^i ← ⊥;
7          for k ∈ dom(v_i) do
8              entry ← M^i[A^i[k]][j];
9              B_{v_i=k} ← v_{i,k} ∧ ⋀_{k'∈(dom(v_i)\{k})} ¬v_{i,k'};
10             B_j^i ← B_j^i ∨ (B_{v_i=k} ∧ B_entry^{i+1});
11         end
12     end
13 end
14 B_∞ ← ⊥;
15 for k ∈ dom(v_1) do
16     entry ← A^1[k];
17     B_{v_1=k} ← v_{i,k} ∧ ⋀_{k'∈(dom(v_1)\{k})} ¬v_{i,k'};
18     B_∞ ← B_∞ ∨ (BDD(v_1 = k) ∧ B_entry^2);
19 end
```

- The final disjunction $B^i_j \vee (B_{v_i=k} \wedge B^{i+1}_{\text{entry}})$ is covered by Theorem A.2 from Appendix A.1: The overall lowest ranked variable (i.e. appearing first in the variable order) is $v_i = 0$, and when restricted to variables $v_i = k$ for $k \in dom(v_i)$ (which are contiguous in the order) the right-hand side represents a complete variable assignment (with $B_{v_i=k}$) that is disjoint to the left-hand side (since we did not consider $v_i = k$ in $B^i_j$ yet).

**Corollary 7.4.** *Given a state $s$ that is detected as a dead-end by a M&S heuristic with* linear *merge strategy (including all PDB heuristics), a dead-end validation for $s$ can be represented by a BDD that can be built in time linear in the size of the cascading tables representation and the maximum domain size in the SAS$^+$ task. This BDD serves as a dead-end validation for* all *dead-ends from this heuristic and must thus only be built once during the entire search.*

This means that a 1-disjunctive BDD certificate can be built and verified for heuristic search using (only) a M&S heuristic with linear merge strategy. For proofs, we can easily show that the BDD $B$ representing all dead-ends is dead, but the mixed statement $\varphi_{\mathbf{R}} \sqsubseteq B$ can be problematic. If $\mathbf{R}$ is MODS, then the statement can be verified; however if $\mathbf{R}$ is a BDD with different variable ordering, the statement cannot be verified. On the other hand, if BDDs are used for the expanded state and we only use one M&S heuristic, we could simplify the proof substantially: Instead of defining sets for each dead-end, we only generate a BDD $E$ with the same variable ordering as $B$ representing all expanded states, and after we show $B$ to be dead we can state $E[A^\Pi] \sqsubseteq (E \cup B)$ directly, since $E$ and $B$ have the same formalism.

## 7.4 Landmarks

Informally speaking, a landmark in planning is something that every plan must achieve along the way. One variation of a landmark is a fact landmark, a variable that must be true in some state visited during the plan. For example, consider a logistics problem where a truck needs to deliver packages. If a package is not already at its goal location, we could define a fact landmark saying that the package must be in the truck at some point. We can also express this as an action landmark, saying that at some point we must load the package into the truck. A landmark heuristic in its simplest form can now determine how far away from a goal a certain state is by counting the landmarks that have not yet been achieved.

An integral part of a landmark heuristic is how landmarks are found. Almost all landmark heuristics described in the literature utilize landmarks detected on the delete relaxation or on the $\Pi^m$ compilation (Keyder, Richter, and Helmert, 2010; Bonet and Helmert, 2010; Bonet and Castillo, 2011). As such, states that are detected as dead-ends by a landmark heuristic are also detected as dead-ends by a delete relaxation or $h^m$ heuristics, and we can use the previous results for generating a dead-end validation for them.

# 8 Other Approaches

## 8.1 Trapper

The *Trapper* algorithm (Lipovetzky, Muise, and Geffner, 2016) is based on the idea of *traps*, which are formulas $\varphi$ such that if a state satisfies $\varphi$, all its successors satisfy it as well. It is easy to see that such traps describe inductive sets.

Trapper is a preprocessing step that finds a special kind of traps called *dead-end traps*, i.e. traps which are inconsistent with the goal. For this, it constructs a graph with nodes $B_i$ representing variable sets and a dummy node $D$. It inserts a labeled edge $a$ from $B_i$ to $B_j$ if for all states $s \supseteq B_i$ where action $a$ is applicable we have $s[a] \supseteq B_j$. If no such $B_j$ exists, an $a$-edge to $D$ is added instead. It then iteratively marks nodes according to certain rules until it reaches a fixpoint. The resulting graph has the following property: If a node is not marked, then for all children reached with the same label at least one of them is not marked as well. Let $\mathcal{T}$ be all the unmarked nodes; then $\varphi_{\text{trap}} = \bigvee_{B_i \in \mathcal{T}} \bigwedge_{v \in B_i} v$ is a trap. The intuition behind this is that if a state contains a set $B_i \in \mathcal{T}$ then for any applicable action the successor contains another $B_j \in \mathcal{T}$.

When building the graph the algorithm additionally uses mutex information $\mathcal{M}$ gained from the $h^2$ heuristic. $\mathcal{M}$ is a collection of tuples that are unreachable when computing $h^2$ for $I^\Pi$. Two variables $x$ and $y$ are called mutex with each other if $\{x, y\} \in \mathcal{M}$. The algorithm only considers variable sets $B_i$ containing at least one variable mutex with a goal variable. Furthermore a set cannot contain variables mutex with each other, and an action is only considered for $B_i$ if its preconditions are not mutex with $B_i$. This means the set represented by $\varphi_{\text{trap}}$ is not necessarily inductive by itself. Instead we need to restrict it to exclude states violating the mutexes. For given mutexes $\mathcal{M}$, $\varphi_{\neg M} = \bigwedge_{m \in M} \bigvee_{v \in m} \neg v$ describes all states consistent with $\mathcal{M}$ (i.e. not violating any mutexes from $\mathcal{M}$). We then have that the set $S$ represented by $\varphi = \varphi_{\text{trap}} \wedge \varphi_{\neg M}$ is inductive. Furthermore $S$ cannot contain any goal states: any state in it must satisfy at least one $B_i \in \mathcal{T}$ but all $B_i$ are mutex with the goal, meaning any state consistent with $B_i$ and $\mathcal{M}$ cannot be a goal state. Finally, if $I^\Pi \in S$, then $S$ is an inductive certificate.

While we have found a formula describing an inductive certificate, we cannot efficiently represent it in its current state: we cannot use BDDs since they cannot efficiently encode $\varphi_{\neg M}$, and $\varphi_{\text{trap}}$ is neither a Horn nor a 2CNF formula. However, we can reformulate $\varphi$ as a disjunction of formulas $\psi_i = (\bigwedge_{v \in B_i} v) \wedge \varphi_{\neg M}$ for each $B_i \in \mathcal{T}$. Each $\psi_i$ is both a Horn and 2CNF formula, and if $\varphi$ represents an inductive certificate, then $\{\psi_i \mid B_i \in \mathcal{T}\}$ represents a 1-disjunctive certificate:

**Theorem 8.1.** *Given a planning task $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$, let $\mathcal{T}$ be a trap obtained from Trapper using mutex information $\mathcal{M}$, and $B_i \subseteq I^\Pi$ for some $B_i \in \mathcal{T}$. Then the set of formulas $\mathcal{F} = \{\psi_i = (\bigwedge_{v \in B_i} v) \land \varphi_{\neg M} \mid B_i \in \mathcal{T}\}$ represents a 1-disjunctive certificate.*

**Proof:** (1) Inclusion of $I^\Pi$: Since $I^\Pi$ is a superset of some $B_i$ and is consistent with $\mathcal{M}$ (thus satisfying $\varphi_{\neg M}$), it must satisfy at least one $\psi_i$.

(2) Exclusion of goal states: For each $\psi_i$ we have that $B_i$ is mutex with the goal, thus no goal state can be a superset of $B_i$ and at the same time satisfy $\varphi_{\neg M}$.

(3) 1-disjunctive inductivity: It is easy to see that the disjunction over the formulas in $\mathcal{F}$ is equivalent to $\varphi_{\text{trap}} \land \varphi_{\neg M}$, which is inductive. For showing 1-disjunctiveness consider any $\psi_i$ and $a \in A^\Pi$: In the graph utilized by the algorithm we know that at least one $a$-successor of $B_i$ must be $B_j \in \mathcal{T}$. This means that for all supersets of $B_i$ consistent with $\mathcal{M}$ and where $a$ is applicable, their successor must be a superset of $B_j$. Since $\psi_i$ denotes exactly the states consistent with $\mathcal{M}$ and being a superset of $B_i$, we have $\mathit{states}(\varphi_i)[a] \subseteq \mathit{states}(\varphi_j)$. $\square$

It is easy to see that each $\psi_i \in \mathcal{F}$ is a Horn formula. Additionally, since mutexes in $\mathcal{M}$ have at most size two because they stem from $h^2$, each $\psi_i$ is also a 2CNF formula.

For our proof system, we can translate the certificate into a proof according to Theorem 5.10. If the proof system contained more set rules, we could also create a proof that requires only formulas $\beta_i = \bigwedge_{v \in B_i} v$ representing each $B_i \in \mathcal{T}$ and $\mu_i = \bigvee_{v \in m_i} \neg v$ representing each mutex $m_i \in \mathcal{M}$. It achieves this by showing that $\mathcal{S} = (\bigcup_{B_i \in \mathcal{T}} \mathit{states}(\beta_i)) \cap \bigcap_{\mu_i \in \mathcal{M}} \mathit{states}(\mu_i)$ is dead and contains the initial state.

We first show that the set of all states violating mutexes, i.e. $\overline{\bigcap_{m_i \in \mathcal{M}} \mathit{states}(\mu_i)}$, is dead. This can be done similar to proving an $h^m$ dead-end dead, only that we do not apply rule **PG** but **PI** which applies if the initial state is in the set and then shows that the complement is dead.

For each $\beta_i$, we know that $\mathit{states}(\beta_i)$ is mutex with the goal, which means we can state $\mathit{states}(\beta_i) \cap \mathit{states}(\mu_j) \cap S_G^\Pi \sqsubseteq \emptyset$ for some $\mu_j$ with **B1**. From this we can show within the proof system that $\mathit{states}(\beta_i) \cap \bigcap_{m_i \in \mathcal{M}} \mathit{states}(\mu_j) \cap S_G^\Pi \sqsubseteq \emptyset$ holds for each $\beta_i$ and thus $\mathcal{S} \cap S_G^\Pi$ is dead.

Next we show that $\mathcal{S}$ leads only to itself or $\overline{\bigcap_{m_i \in \mathcal{M}} \mathit{states}(\mu_i)}$ (which is dead). We iterate over all $\beta_i$ and $a \in A^\Pi$. If $\mathit{pre}(a)$ is mutex with $B_i$, we can state $(\mathit{states}(\beta_i) \cap \mathit{states}(\mu_j))[a] \sqsubseteq \emptyset$ for some $\mu_j$ with **B2**. Otherwise we can state $\mathit{states}(\beta_i)[a] \sqsubseteq \mathit{states}(\beta_j)$ for some $\beta_j$ with **B2**. In either case we can show within the proof system that judgment $\mathit{states}(\beta_i)[a] \cap \bigcap_{m_i \in \mathcal{M}} \mu_i \sqsubseteq \bigcup_{B_i \in \mathcal{T}} \beta_i$ holds. From these judgments we can derive $\mathcal{S}[A^\Pi] \sqsubseteq \bigcup_{B_i \in \mathcal{T}} \beta_i$. At this point the current proof system lacks an axiom of the form $X \subseteq (X \cap Y) \cup \overline{Y}$, or a derivation thereof. If we had this judgment, the claim at the beginning of the paragraph would follow directly. Since $\mathcal{S}$ only leads to itself and dead-ends and contains only dead goal states, we can thus show that $\mathcal{S}$ is dead.

It is left to show that $\{I^\Pi\} \sqsubseteq \mathcal{S}$ holds, which can be done by stating with **B1** that $\{I^\Pi\} \sqsubseteq \mathit{states}(\beta_i)$ for some $\beta_i$ and $\{I^\Pi\} \sqsubseteq \mathit{states}(\mu_i)$ for all $\mu_i$. The desired judgment can then be derived within the proof system.

The advantage of constructing the proof this way rather than directly translating the 1-disjunctive certificate is that the formulas involved can be represented more compactly (without having basic statements over a large amount of set variables), especially since we do not need to repeat the mutex information. It also follows the idea of Trapper more naturally, since it first proves that the mutex information is correct and then shows that *with background knowledge of the mutexes* the trap is a trap. The disadvantage is that the proof might require substantially more steps. This should not affect performance too much since rule applications are fast to verify, but it will affect the size of the written proof.

While we mostly focused on the case where the trap found by Trapper contains the initial state, Trapper can also be used in a planning system to prune the search space for problems where the initial state is not part of the trap. In these cases it is unlikely that we can build an inductive certificate due to their lack of composability. The proof system on the other hand can easily combine the knowledge that all states in the trap are dead with knowledge from the other planning component, and if need be can even derive more fine grained statements about which states are dead, for example we could derive that $\textit{states}(\beta_i) \cap \bigcap_{m_i \in \mathcal{M}} \textit{states}(\mu_i)$ is dead for some specific $B_i$.

## 8.2 Iterative Dead Pairs Calculation

Alcázar and Torralba (2015) describe an algorithm that finds pairs of SAS$^+$ facts $\{p, q\}$ such that any state satisfying $p \wedge q$ is dead. In the STRIPS setting, the most faithful adaptation of the algorithm considers pairs of *literals*, i.e., $p$ and $q$ can be state variables or negated state variables. Such fact pairs are sometimes called "forward/backward mutexes", but since they are not mutexes in the strict sense, we call them *dead pairs* instead. A *dead pair set* $D$ is a set of dead pairs. A state is *consistent with* $D$ if it contains no pair in $D$; otherwise it is *pruned* by $D$. Pruned states are dead.

The algorithm by Alcázar and Torralba alternately performs forward and backward steps. The $k$-th step computes a dead pair set $D_k$, exploiting that $D_{k-1}$ is already a known dead pair set (beginning with $D_0 = \emptyset$). If the $k$-th iteration is forward, the algorithm performs an $h^2$-style reachability analysis using $D_{k-1}$ as background knowledge, i.e., ignoring states pruned by $D_{k-1}$. The backward iterations are similar, but using a backward $h^2$-style analysis.

In a *forward* step, the new dead pair set $D_k$ is the (uniquely defined) maximal set of pairs such that: (a) if $I^\Pi$ is consistent with $D_{k-1}$, then $I^\Pi$ is consistent with $D_k$, and (b) for all transitions $s \rightarrow s'$ where $s$ is consistent with $D_k$ and $s'$ is consistent with $D_{k-1}$, $s'$ is consistent with $D_k$. This characterization is not apparent from the description by Alcázar and Torralba, but it follows directly from the description of Rintanen's (2008) invariant synthesis algorithm, which is equivalent to $h^2$. Note that in a STRIPS setting the first forward iteration will find all mutex information encoded in the multi-valued variables of a SAS$^+$ task.

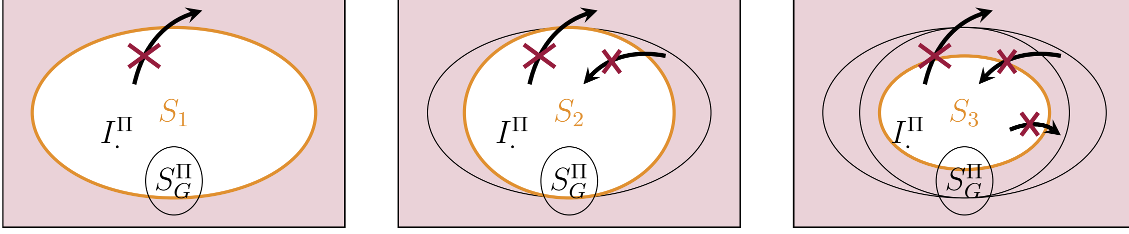



Figure 8.1: An example of an iterative dead pairs calculation.

Similarly, in a *backward* step, the new dead pair set $D_k$ is the (uniquely defined) maximal set of pairs such that: (a') all goal states consistent with $D_{k-1}$ are consistent with $D_k$, and (b') for all transitions $s \to s'$ where $s'$ is consistent with $D_k$ and $s$ is consistent with $D_{k-1}$, $s$ is consistent with $D_k$.

An example of how the algorithm works can be seen in Figure 8.1. In the first iteration, a simple inductive set $S_1$ is found which denotes the states containing only tuples that were reachable by the $h^2$ calculation for the initial state. In the second iteration, a backwards analysis is performed from the goal states in $S_1$, discovering new states as dead. Those states might possibly have been backwards-reachable from some goal state, but only via a state in $\overline{S_1}$ and are thus not reached by the backwards analysis. In the first step a forward analysis is run again, but this time some states that were previously reachable are not reachable anymore, since they were reached over states in $\overline{S_2}$.

We can construct a proof in our proof system showing that all states pruned by $D_k$ are dead. Let $S_k$ be the set of states consistent with $D_k$, which can be described by the 2CNF formula $\bigwedge_{\{\ell_1,\ell_2\}\in D_k}(\overline{\ell_1} \vee \overline{\ell_2})$, where $\overline{\ell}$ denotes the complement of the literal $\ell$. We prove that all sets $\overline{S_k}$ are dead. For $k = 0$, this is easily proven by stating **B1** $\overline{S_k} \sqsubseteq \emptyset$ and applying **SD** together with **ED**.

For $k > 0$, we have already proven (1): $\overline{S_{k-1}}$ is dead. In a forward step, we can assume $I^\Pi \notin \overline{S_{k-1}}$ or else we could have proven unsolvability in the previous step. Hence we can use the basic statement **B1** (2): $\{I^\Pi\} \subseteq S_k$ (**B1**). From (b) we get the basic statement **B2** (3): $S_k[A] \subseteq S_k \cup \overline{S_{k-1}}$. Using (3), (1), (2) with **PI**, we derive that $\overline{S_k}$ is dead as required. If at this point $\overline{S_k}$ contains all goal states, we can state $S_G^\Pi \sqsubseteq \overline{S_k}$ and derive with **SD** that $S_G^\Pi$ is dead, which allows us to conclude the proof with **CG**.

In a backward step, (a') can be rephrased as "all goal states not consistent with $D_k$ are not consistent with $D_{k-1}$", yielding the basic statement **B1**: $\overline{S_k} \cap S_G^\Pi \subseteq \overline{S_{k-1}}$, which according to **SD** together with (1) proves (2'): $\overline{S_k} \cap S_G^\Pi$ is dead. From (b') we obtain the basic statement **B3** (3'): $[A]S_k \subseteq S_k \cup \overline{S_{k-1}}$. Using (3'), (1), (2') in **RG**, we derive that $\overline{S_k}$ is dead as required. If at this point $I^\Pi$ is in $\overline{S_k}$, we can conclude the proof by stating $\{I^\Pi\} \sqsubseteq \overline{S_k}$, from this deriving that $\{I^\Pi\}$ is dead with **SD** and finally applying **CI**.

In summary, we can generate a compact proof in our proof system showing that all states that are prunable according to the iterative dead pairs algorithm are dead. Using **SD**, it is also easy to extract fine-grained results of the form "All states satisfying $\ell_1 \wedge \ell_2$ are dead" for each dead pair $\{\ell_1, \ell_2\}$, which can be converted to representations other than

2CNF (e.g. BDDs or Horn clauses) and used in a larger overall proof. For example, it is not difficult to augment a proof of unsolvability for explicit search to include pruning of states that satisfy a dead pair. We conjecture that it is also possible to extend proofs of unsolvability for certain heuristics to take such dead pairs into account as in *constrained abstraction* (Haslum, Bonet, and Geffner, 2005).

## 8.3 Clause-Learning State Space Search

Steinmetz and Hoffmann (2017) describe a search algorithm built upon the $h^C$ heuristic family where the heuristic is incrementally improved to detect more dead-ends. The algorithm performs a depth-first search where states are pruned by an increasingly stronger variant of the $h^C$ heuristic. Another way to see this is that the depth-first search prunes states based on a *family* of $h^C$ heuristics. Thus, we can generate a witness if we can find a dead-end validation for each such state pruned by one of the heuristics.

The heuristic $h^C$ is parameterized by a set $C$, where each $c \in C$ is a set of state variables (Keyder, Hoffmann, and Haslum, 2014; Steinmetz and Hoffmann, 2017). It is a generalization of the $h^m$ heuristic family, which considers *all* sets up to size $m$. As such, a compilation $\Pi^C$ (Haslum, 2012) analogous to $\Pi^m$ can be generated and we can show the same way as we did for $h^m$ that the set $U_s^C$ of variable sets unreachable in $\Pi^C$ from $s$ can be used to define an inductive set:

**Proposition 8.1.** *Given a task* $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$ *and a state* $s$*, the intersection of the family of sets* $\mathcal{F} = \{S_{\bar{c}} \mid c \in U_s^C\}$ *is inductive, where* $S_{\bar{c}} = \{s \subseteq V^\Pi \mid c \nsubseteq s\}$ *and* $U_s^C$ *is the set of unreachable variable sets from* $s$ *determined by* $h^C$.

As with $h^m$ we can thus generate a 1-disjunctive certificate represented as a Horn formula. In the proof system we can cover the algorithm as described in Chapter 6 where the sub-proof for each dead-end uses Horn formulas as representation.

The search algorithm can also be set to continue improving its current $h^C$ heuristic even after the search space has been explored, up to the point where it is strong enough to detect the initial state as dead. In this case the dead-end validation directly serves as a 1-conjunctive certificate.

One enhancement of clause-learning state space search is that it can learn clauses with the property that any state not satisfying the clause is guaranteed to be detected as dead-end by the current $h^C$ heuristic. Each state is then first tested, and if it does not satisfy the clause it can be pruned without evaluating $h^C$ on it. For generating dead-end validations however, we must evaluate each dead-end $d$ with $h^C$ in order to obtain $U_d^C$ and cannot profit from this enhancement.

# Part III

# Experimental Evaluation

# 9 Inductive Certificates

In order to assess how suitable inductive certificates are in practice, we augmented the Fast Downward planning system to generate 1-disjunctive BDD certificates for $A^*$ search combined with a relaxation heuristic like $h^{\max}$ or $h^{\text{add}}$, or with a M&S heuristic with linear merge strategy. Furthermore we implemented a verifier capable of handling simple inductive BDD certificates as well as $r$-disjunctive and $r$-conjunctive BDD certificates, using the CUDD library (Somenzi, 2015) for representing and manipulating BDDs. Both implementations are publicly available (Eriksson, 2019a).

We tested generation of certificates and subsequent verification through our verifier on the benchmark set detailed in Chapter 1.4 for the following configurations:

- $h^{\max}$: $A^*$ search guided by the $h^{\max}$ heuristic.
- $h^{\text{M\&S}}$: $A^*$ search guided by the $h^{\text{M\&S}}$ heuristic with the following setting:
  - `merge_strategy=merge_precomputed(`
    `merge_tree=linear())`
    (enforces a linear merge strategy)
  - `shrink_strategy=shrink_bisimulation()`
  - `label_reduction=exact(`
    `before_shrinking=true,before_merging=false)`
  - `prune_unreachable_states=false`
    (ensures inductivity, see Chapter 7.3)

We also ran an unaltered version of Fast Downward to analyze the overhead induced by certificate generation. We call the unaltered version FD and the version that generates certificates $\text{FD}^{\text{C}}$. The verifier for inductive certificates is called $\text{Ver}^{\text{C}}$.

Table 9.1 gives a general overview of the results by showing how many tasks FD and $\text{FD}^{\text{C}}$ could solve within 30 minutes and 3584 MiB memory, and how many certificates our verifier could verify within 4 hours and 3584 MiB memory. Overall, in 82% of the cases where FD could solve the task, $\text{FD}^{\text{C}}$ could solve and additionally generate a certificate within the same limits; and 87% of the generated certificates could be verified within the given limits.

In the following sections, we first describe the most important aspects of our implementation, then analyze the most significant reasons for overhead when generating the certificates, and afterwards discuss the performance of the verifier.

| | $h^{max}$ | | | $h^{M\&S}$ | | |
|---|---|---|---|---|---|---|
| | FD | $FD^C$ | $Ver^C$ | FD | $FD^C$ | $Ver^C$ |
| 3unsat (30) | 15 | 10 | 10 | 15 | 10 | 10 |
| bag-barman (20) | 8 | 8 | 4 | 11 | 8 | 4 |
| bag-gripper (25) | 2 | 2 | 2 | 3 | 2 | 2 |
| bag-transport (29) | 7 | 6 | 5 | 7 | 5 | 4 |
| bottleneck (25) | 21 | 17 | 15 | 10 | 8 | 8 |
| cave-diving (25) | 7 | 7 | 6 | 7 | 7 | 6 |
| chessboard-pebbling (23) | 5 | 4 | 4 | 5 | 4 | 4 |
| diagnosis (13) | 5 | 5 | 5 | 4 | 4 | 4 |
| document-transfer (20) | 7 | 6 | 6 | 12 | 12 | 12 |
| mystery (9) | 2 | 2 | 1 | 2 | 1 | 1 |
| nomystery (150+24) | 59 | 38 | 25 | 60 | 45 | 37 |
| pegsol (24) | 24 | 24 | 24 | 24 | 24 | 24 |
| pegsol-row5 (15) | 5 | 4 | 4 | 5 | 4 | 4 |
| rovers (150+20) | 15 | 12 | 8 | 25 | 24 | 23 |
| sliding-tiles (20) | 10 | 10 | 10 | 10 | 10 | 10 |
| tetris (20) | 10 | 5 | 5 | 5 | 5 | 5 |
| tpp (25+30) | 23 | 15 | 12 | 37 | 34 | 29 |
| total (697) | 225 | 175 | 146 | 242 | 207 | 187 |

Table 9.1: Completed tasks by domain for Fast Downward, certifying Fast Downward and certificate verification.

## 9.1 Implementation

In order to generate a 1-disjunctive BDD certificate, $FD^C$ must create the following BDDs:

- One BDD for each expanded state representing only this state.
- A set of BDDs that represent inductive sets without goal states, such that each dead-end is contained in at least one BDD. Several dead-ends may be contained in the same BDD.

In order to improve efficiency when verifying the certificate, we also emit a *hint* file, which denotes for all BDDs representing expanded states and all actions applicable to that state which BDD contains the successor state. This reduces the inductivity check from time quadratic in the size of the certificate to linear, as it does not need to test all BDDs in order to find the correct successor BDD anymore. Since the number of BDDs in our certificate can grow very large, a reduction from quadratic to linear time impacts performance greatly. Due to the hint file, each BDD in the certificate is associated with a unique `setid`.

**A$^*$ search** The search engine is responsible for creating a BDD for each expanded state, creating the hint file and requesting a BDD from the heuristic for each dead-end. The hint

file is created during the search for each expansion. If the heuristic reports a successor to be a dead-end, the search requires a `setid` from the heuristic for the dead-end. Otherwise, the `stateid` provided by Fast Downward is used. After search is concluded, we iterate over all generated states and create the actual BDDs for each expanded state. We do not do this during the search in order to impact it as little as possible; this way the planner can still answer the question on whether the problem is solvable or not even when creating a certificate requires too much memory or time. Finally we write all BDDs representing expanded states into a file, and request the heuristic to do the same with its BDDs.

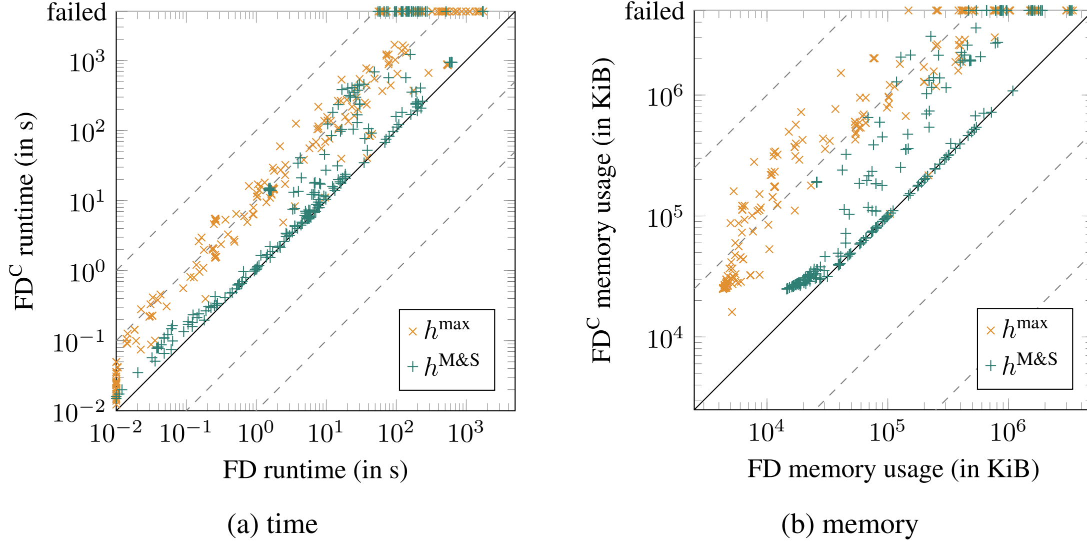
**$h^{\mathbf{max}}$ heuristic** The $h^{\max}$ heuristic generates a new BDD each time it evaluates a state as dead-end during search, and uses its `stateid` as `setid`. While we could reduce search overhead further by only creating the BDDs after search has concluded, it would lead to more total overhead as the heuristic must be evaluated again in order to get the set of unreachable variables.

**$h^{\mathbf{M\&S}}$ heuristic** Unlike $h^{\max}$ the $h^{\mathrm{M\&S}}$ heuristic generates one BDD representing all dead-ends only after search concluded and if at least one dead-end was encountered. This is possible since only the M&S representation is needed for constructing the BDD, which is saved until the program exits. During search only the `stateid` of the first encountered dead-end is stored, which is used as `setid` for the BDD.

**Verifier** After reading in the certificate, the verifier tests the three properties of an inductive certificate. For testing inductivity of $r$-conjunctive or $r$-disjunctive certificates, a boolean is stored for each set indicating if the checks for this particular set have been performed already. If a hint file is present, the checks related to the particular hint are processed as the hint is read, eliminating the need to store the hints. Afterwards, all sets whose checks have not been performed yet are (for each action) first tested whether they are inductive themselves, and if not whether we can find a combination of $r$ sets satisfying the needed property. For certificates created by $\mathrm{FD}^{\mathrm{C}}$ this guarantees a linear time complexity in the number of sets and actions: for each set representing an expanded state and each applicable action the corresponding set is detailed in the hint file while for each inapplicable action the progression is empty and thus a subset of the set itself; and each set created from a dead-end is inductive.

## 9.2 Generation

Generating a certificate succeeded in most cases: for $h^{\max}$ in 78% of all cases and for $h^{\mathrm{M\&S}}$ in 86%. The size of the created certificates range from a few KiB to a little over 8.5 GiB, with an average of approximately 530 MiB, and certificates for $h^{\mathrm{M\&S}}$ typically being moderately smaller.




(a) time

(b) memory

Figure 9.1: Comparison of time needed and memory used between FD and FD$^C$. Only tasks that FD completed are shown.

| | $h^{max}$ | | $h^{M\&S}$ | |
|---|---|---|---|---|
| | memory | time | memory | time |
| FD$^C$ | -12 | -38 | -32 | -3 |
| FD$^{C'}$ | -5 | -46 | 0 | -28 |

Table 9.2: Reasons for failure to generate certificates in tasks that FD solved.

Figure 9.1 compares time and memory used for FD and FD$^C$. We can see that creating certificates induces a significant overhead for $h^{max}$ but in most cases only moderate overhead for $h^{M\&S}$. This is due to the fact that M&S often spends a significant amount of allotted time and memory for calculating its abstraction, a process which is not altered by FD$^C$. Search with $h^{max}$ on the other hand spends almost all of its time expanding states since $h^{max}$ does not need any preprocessing and is in general fast to evaluate. Furthermore, $h^{max}$ needs to calculate and store a BDD for each encountered dead-end, while $h^{M\&S}$ only needs to store one. While this BDD is usually significantly larger than a BDD for a $h^{max}$ dead-end, it is proportional to the M&S representation and takes less time and memory to calculate than the M&S representation itself.

The first row in Table 9.2 shows why FD$^C$ failed to create a certificate where FD solved the task within limits. $h^{max}$ mostly fails due to the time limit, while $h^{M\&S}$ almost exclusively fails due to the memory limit. The tasks where $h^{M\&S}$ induces significant memory overhead all required a significant amount of expansions, thus much memory is needed to store the BDDs of these expanded states. In order to improve memory usage, we implemented an alternative version FD$^{C'}$, which does not store any BDDs for expanded states but instead directly writes them into a text file, without even calling the CUDD library.

| | $h^{\max}$ | | | $h^{\text{M\&S}}$ | | |
|---|---|---|---|---|---|---|
| generation | finished/#certificates | memory | time | finished/#certificates | memory | time |
| $\text{FD}^{\text{C}}$ | 146/175 | 0 | -29 | 187/207 | -1 | -19 |
| $\text{FD}^{\text{C}'}$ | 144/174 | -1 | -29 | 189/214 | -6 | -19 |
| $\text{FD}^{\text{C}''}$ | 66/175 | 0 | -109 | 114/207 | -1 | -92 |

Table 9.3: Reasons for failure of $\text{Ver}^{\text{C}}$. "finished" denotes the number of tasks where $\text{Ver}^{\text{C}}$ could verify the certificate within the given limits.

As we can see in the second row of Table 9.2, $\text{FD}^{\text{C}'}$ is able to improve coverage significantly for M&S, shifting the reason for failure compared to FD fully to a reached time limit, and reducing the average (max) ratio of memory used compared to FD from 2.452 (16.072) to 1.006 (1.237).[1]

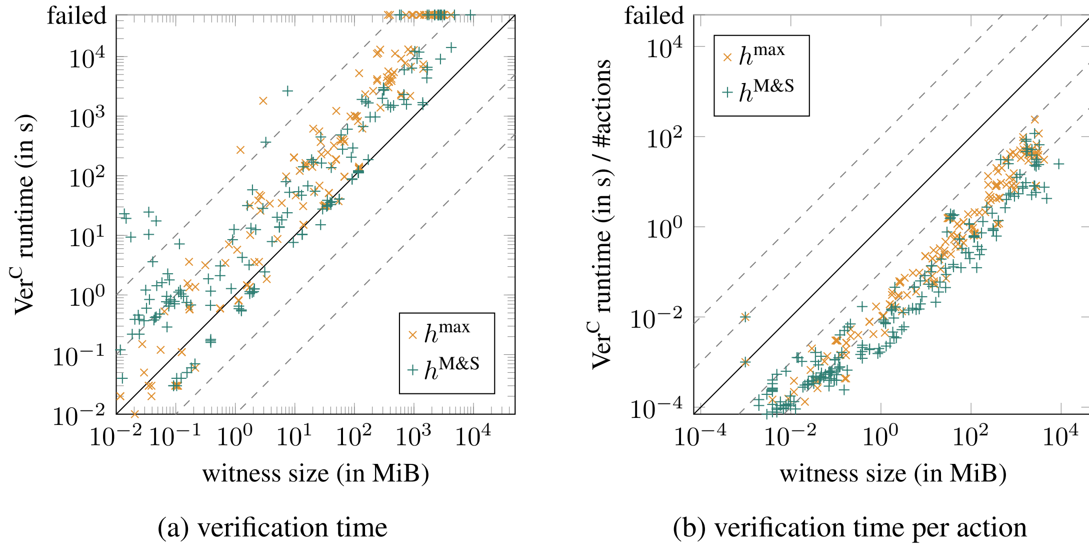
This improvement comes with a price however. The certificates generated by $\text{FD}^{\text{C}'}$ are on average over twice as big as those generated by $\text{FD}^{\text{C}}$, with a worst case factor of over 12. The reason for this is that when storing the BDDs with CUDD, nodes can be shared between BDDs, reducing overall size. Additionally, all meta information (such as amount of variables) must be repeated for each single BDD in $\text{FD}^{\text{C}'}$, while it is only written once in $\text{FD}^{\text{C}}$.

## 9.3 Verification

The certificates generated by $\text{FD}^{\text{C}}$ could be verified in 83% of all cases for $h^{\max}$ and 90% for $h^{\text{M\&S}}$ within limits; with all verified certificates being valid. The first row in Table 9.3 shows that all but one certificate failed due to the time limit. One explanation is that the verifier needs most of its memory to store the BDDs belonging to the certificate. Otherwise it only needs to store BDDs representing the actions and one temporary BDD at a time in order to verify the properties of the certificate. Since the search needed to be able to store all BDDs (on top of its normal memory requirement) within the same memory limit in order to even create the certificate, it is reasonable to assume that the verifier should be able to verify the certificate with the same memory limit. Another factor is however that CUDD dynamically allocates memory for a cache and deletes BDDs lazily. Thus, when memory reaches exhaustion, CUDD slows down significantly in order to free as much memory as possible, which can then lead to a timeout.

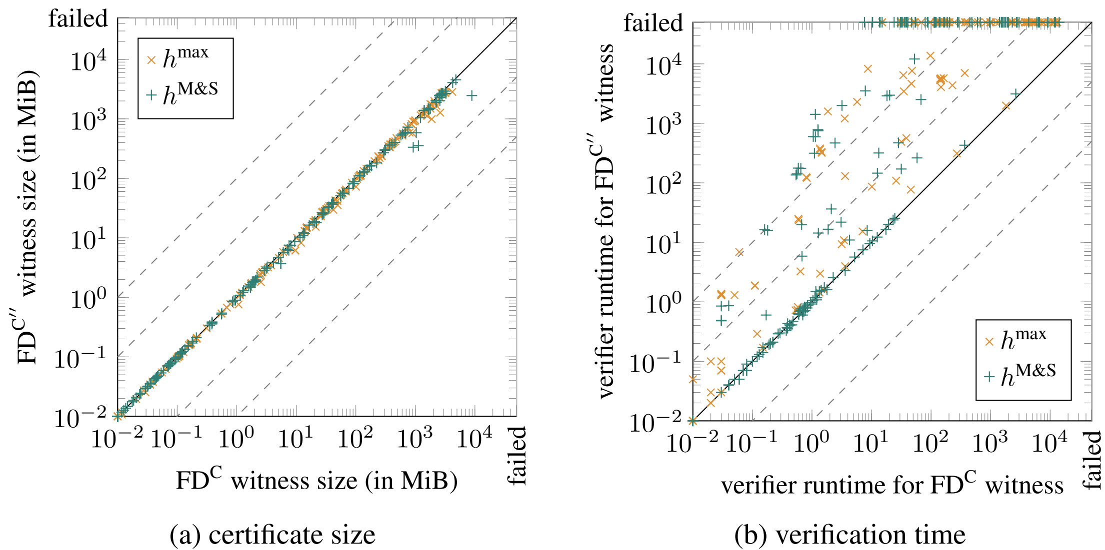
Figure 9.2 (a) depicts the time needed to verify the certificate as a function of the certificate size, showing that verification time is roughly linear in certificate size, although bigger certificates typically cause more overhead. However, we also need to take into account that larger certificates typically stem from tasks with more actions and facts, which both also impact verification time. While the amount of facts is factored into the certificate size since BDDs for more facts are bigger, actions are only factored in through

[1]Table B.1 in Appendix B shows full coverage results for $\text{FD}^{\text{C}'}$.




(a) verification time (b) verification time per action

Figure 9.2: Time needed for verification of the certificate as a function of its size. Only tasks for which a certificate could be generated are shown.




(a) certificate size (b) verification time

Figure 9.3: Comparison between certificates from $\mathrm{FD}^{\mathrm{C}}$ and $\mathrm{FD}^{\mathrm{C}''}$. Tasks for which neither method could create a certificate are omitted.

the hint file, which is usually only a small factor in the overall certificate size. If we factor in actions by dividing the verification time by the amount of actions as shown in (b), the linear relationship becomes more visible.

Comparing $\mathrm{FD}^{\mathrm{C}}$ and $\mathrm{FD}^{\mathrm{C}'}$, Table 9.3 shows that $\mathrm{FD}^{\mathrm{C}'}$ is able to verify roughly the same amount of certificates as $\mathrm{FD}^{\mathrm{C}}$. The tasks for which $\mathrm{FD}^{\mathrm{C}'}$ could create certificates but not $\mathrm{FD}^{\mathrm{C}}$ mostly failed verification due to the memory limit. Since $\mathrm{FD}^{\mathrm{C}'}$ does not need to store the entire certificate while creating it and as discussed above the verifier does not require much additional memory other than storing the certificate, these results suggest that those cases where $\mathrm{FD}^{\mathrm{C}'}$ succeeds but $\mathrm{FD}^{\mathrm{C}}$ does not yield certificates that require more memory to store than is allotted in the experiment.

To show the importance of the hint file, we ran a version $\mathrm{FD}^{\mathrm{C}''}$ where no hint file is created. Figure 9.3 compares the certificate size and the verifier runtime of $\mathrm{FD}^{\mathrm{C}}$ and $\mathrm{FD}^{\mathrm{C}''}$. As we can see the hint file often does not contribute much to the total certificate size, but verification without the hint file is significantly slower. As a result, $\mathrm{FD}^{\mathrm{C}''}$ fails considerably more often, as shown in the last row of Table 9.3, and almost exclusively due to a reached time limit.[2]

[2]Table B.2 in Appendix B shows full coverage results for $\mathrm{FD}^{\mathrm{C}''}$.

# 10 Proof System

For the experimental evaluation of the proof system, we augmented Fast Downward to create proofs for A* search when using the $h^{\text{M\&S}}$, $h^m$ or a delete relaxation heuristic. Additionally, we showcase the combination of different information sources by adding proof generation support to the maximum evaluator, which takes several heuristics and uses the maximum of their estimates as heuristic value. Finally, we augment the implementation of the $h^C$-based clause-learning algorithm from Steinmetz and Hoffmann (2017) to emit proofs for the case where the refinement of the $h^C$ heuristic is continued until the initial state is detected as dead-end. To verify the generated proofs, we implemented a verifier with support for the four formalisms considered in this thesis (BDDs, Horn formulas, 2CNF formulas and MODS). All of our implementations are publicly available (Eriksson, 2019a). As in the previous chapter our implementation uses the CUDD library (Somenzi, 2015) for representing and manipulating BDDs, and the experiments were run on the benchmark set described in Chapter 1.4.

For the experimental evaluation we used the following configurations:

- $h^{\max}$: A* search guided by the $h^{\max}$ heuristic.
- $h^{\text{M\&S}}$: A* search guided by the $h^{\text{M\&S}}$ heuristic with the following setting:
  - `merge_strategy=merge_precomputed(merge_tree=linear())` (enforces a linear merge strategy)
  - `shrink_strategy=shrink_bisimulation()`
  - `label_reduction=exact(before_shrinking=true,before_merging=false)`
  - `prune_unreachable_states=false` (ensures inductivity, see Chapter 7.3)
- $h^2$: A* search guided by the $h^m$ heuristic with $m = 2$.
- *max*($h^{\text{M\&S}}, h^2$): A* search guided by the maximum of the $h^2$ and $h^{\text{M\&S}}$ heuristics (parameters as above)
- $h^C$: Clause Learning State Space Search(Steinmetz and Hoffmann, 2017) with the following setting:
  - `--heuristic "hff=ff()"`

| | $h^{\max}$ | | | $h^{\mathrm{M\&S}}$ | | | $h^2$ | $max(h^{\mathrm{M\&S}}, h^2)$ | $h^C$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | FD | $\mathrm{FD^P}$ | $\mathrm{Ver^P}$ | FD | $\mathrm{FD^P}$ | $\mathrm{Ver^P}$ | all | all | CLS | $\mathrm{CLS^P}$ | $\mathrm{Ver^P}$ |
| 3unsat (30) | 15 | 10 | 10 | 15 | 10 | 10 | 5 | 5 | 5 | 5 | 5 |
| bag-barman (20) | 8 | 8 | 8 | 11 | 8 | 8 | 0 | 0 | 0 | 0 | 0 |
| bag-gripper (25) | 2 | 2 | 2 | 3 | 3 | 3 | 0 | 0 | 0 | 0 | 0 |
| bag-transport (29) | 7 | 6 | 6 | 7 | 6 | 6 | 15 | 10 | 2 | 2 | 2 |
| bottleneck (25) | 21 | 17 | 15 | 10 | 9 | 9 | 12 | 12 | 9 | 9 | 9 |
| cave-diving (25) | 7 | 7 | 7 | 7 | 7 | 7 | 2 | 2 | 6 | 6 | 6 |
| chessboard-pebbling (23) | 5 | 5 | 5 | 5 | 5 | 5 | 2 | 2 | 2 | 2 | 2 |
| diagnosis (13) | 5 | 5 | 5 | 4 | 4 | 4 | 2 | 2 | 8 | 8 | 8 |
| document-transfer (20) | 7 | 6 | 6 | 12 | 12 | 12 | 2 | 8 | 5 | 5 | 4 |
| mystery (9) | 2 | 1 | 1 | 2 | 2 | 1 | 8 | 8 | 7 | 7 | 7 |
| nomystery (150+24) | 59 | 35 | 26 | 60 | 45 | 37 | 45 | 54 | 139 | 138 | 138 |
| pegsol (24) | 24 | 24 | 24 | 24 | 24 | 24 | 14 | 14 | 14 | 14 | 14 |
| pegsol-row5 (15) | 5 | 5 | 5 | 5 | 5 | 5 | 3 | 3 | 4 | 4 | 4 |
| rovers (150+20) | 15 | 11 | 9 | 25 | 24 | 23 | 66 | 70 | 155 | 155 | 155 |
| sliding-tiles (20) | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | 0 | 0 | 0 |
| tetris (20) | 10 | 5 | 5 | 5 | 5 | 5 | 0 | 5 | 0 | 0 | 0 |
| tpp (25+30) | 23 | 15 | 12 | 37 | 33 | 29 | 9 | 14 | 32 | 33 | 33 |
| total (697) | 225 | 172 | 156 | 242 | 212 | 198 | 185 | 209 | 388 | 388 | 387 |

Table 10.1: Completed tasks by domain for the original planner (FD and CLS), the proof generating planner ($\mathrm{FD^P}$ and $\mathrm{CLS^P}$) and proof verification ($\mathrm{Ver^P}$). For $h^2$ and $max(h^{\mathrm{M\&S}}, h^2)$, all tasks completed by FD were also completed by $\mathrm{FD^P}$ and $\mathrm{Ver^P}$; we thus summarize the respective numbers.

- `--heuristic "u=uc(x=-1,clauses=statemin)"`
- `--search "dfs(eval=[hff], preferred=[hff], refiner=[ucrn2_1(uc=u)], u_refine_initial_state=true)"`
  (the last parameter enforces refinement until $I^\Pi$ is detected as dead-end).

As with inductive certificates, we compare our augmented version of Fast Downward $\mathrm{FD^P}$ against FD in order to assess the overhead from generating proofs, and do the same for proof generating Clause Learning State Space Search $\mathrm{CLS^P}$ and its original version CLS. Table 10.1 summarizes on how many problem the original planner and the proof generating planner successfully terminated within 30 minutes and 3584 MiB memory, and how many proofs the verifier could verify within the 4 hours and 3584 MiB memory. In total we could generate proofs in 93% of all cases where the original planner finished within limits, and could verify 97% of all generated proofs.

## 10.1 Implementation

A proof for heuristic search first proves for all dead-ends that they are dead. We then build a set of all dead-ends which we show to be dead because all its members are dead, and a set containing all expanded states which is dead because its successor states are either contained in it or dead. Finally, we show that the task is unsolvable by showing that $I^\Pi$ is dead because it is in the set of expanded (or dead-end) states.

Our implementation writes the proof only after search terminated. The search engine then iterates over all generated states $s$. If $s$ has been expanded it adds the state to a BDD $B_{\text{exp}}$ which will represent all expanded states. If $s$ has been recognized as dead-end, we ask the heuristic to write a sub-proof showing that some set containing $s$ is dead. We then define an explicit set $E_s$ which only contains $s$, and show that it is dead because it is a subset of the set shown dead by the heuristic. We also perform the proof steps needed to ultimately show that the union of all $E_s$ is dead and build both an explicit set $E_{\text{dead}}$ and a BDD $B_{\text{dead}}$ containing all dead-ends. Both sets are needed in order to translate between MODS and BDD formalisms since mixed representation statements only allow one set on each side. This way, $E_{\text{dead}}$ can be shown to be a subset of the union of all $E_s$, and $B_{\text{dead}}$ can then shown to be a subset of the singular set $E_{\text{dead}}$. While we could also forgo creating $B_{\text{dead}}$ by saving the expanded states to an explicit set as well, we decided against it since the verifier needs to calculate the progression of this set, which can be significantly faster for BDDs.

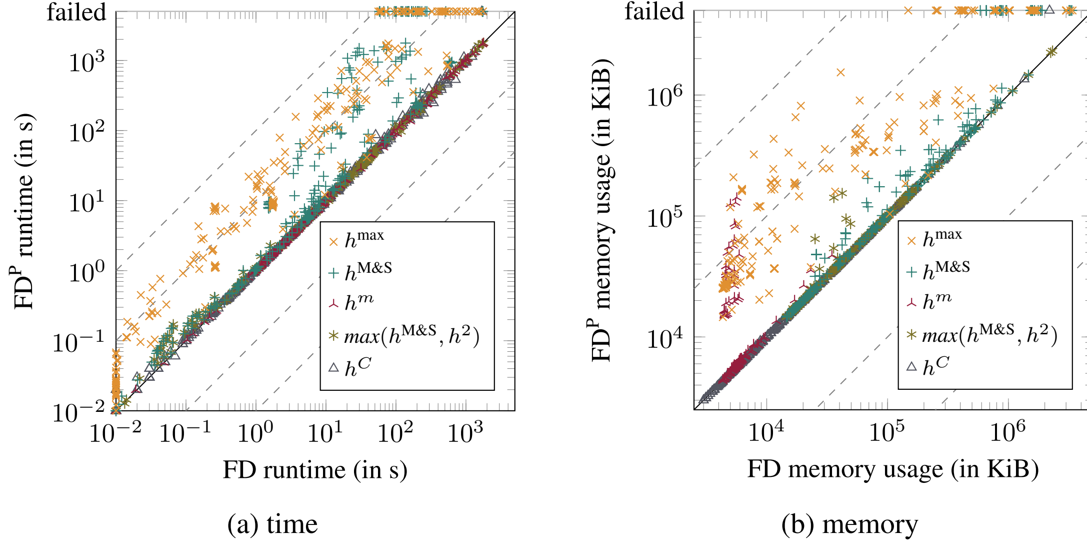
The heuristics have the possibility of saving information about dead-ends during the search, such that we can avoid recomputing the heuristic when writing the proof. $h^{\text{max}}$ and $h^m$ both use this possibility, saving the unreachable (tuples of) propositions for each dead-end encountered. In terms of used formalisms, $h^{\text{max}}$ and $h^{\text{M\&S}}$ use BDDs for their sub-proofs, while $h^m$ uses Horn formulas.

In the proof, each set (both concrete and compound sets) and each judgment is associated with an ID, which must be declared in ascending order. The verifier reads in the proof line by line, checks directly whether the denoted judgment is correct and if so adds it to its knowledge base. If the verification of a judgment fails, the verifier prints an error message but continues its execution. Similarly, if the proof continues past the point of proving unsolvability, the verifier checks the following judgments as well. When terminating it simply outputs whether at some point it derived the judgment “task unsolvable”. This architecture allows for a slightly more general usage, since the verifier is not only focused on proving unsolvability.

## 10.2 Generation

We first look into the tasks that the original planner detected as unsolvable within limits but the proof generating planner failed. Across all configurations, only 5 tasks in $h^{\text{max}}$ failed due to a reached memory limit, while failing due to a timeout happens in 78 tasks across $h^{\text{max}}$ and $h^{\text{M\&S}}$. For configurations $h^2$, $\mathit{max}(h^{\text{M\&S}}, h^2)$ and $h^C$, no tasks were lost.

Figure 10.1 compares the time and memory usage for the original planner versus the proof generating planner. Regarding time usage, we see that generating proofs does not impact the planner noticeably when using $h^2$, $\mathit{max}(h^{\text{M\&S}}, h^2)$ and $h^C$, but incurs significant overhead with the configurations $h^{\text{max}}$ and $h^{\text{M\&S}}$. For $h^{\text{max}}$, this happens almost always, while for $h^{\text{M\&S}}$ the results are more diverse and significant overhead only occurs on larger tasks.




(a) time (b) memory

Figure 10.1: Comparison of time needed and memory used between FD and FD$^P$. Only tasks that FD completed are shown.

For configurations other than $h^C$, these results can be explained when considering the number of new states discovered per second. A proof needs to incorporate each state discovered during search, either by adding it to the BDD containing all expanded states, or by writing a deadness proof (and building the proof that all dead-ends are dead). $h^2$ and $max(h^{M\&S}, h^2)$ both discover new states at a very slow rate since calculating $h^2$ is expensive. $h^{M\&S}$ has a long preprocessing phase, but is afterwards very fast since it only needs to perform a look-up in order to obtain the heuristic value of a state. Finally, $h^{max}$ expands states at a very high rate because it is fast to calculate and spends the entire time expanding states since it does not need any preprocessing.

$h^C$ is different in this aspect, since it is run until its heuristic can detect the initial state as a dead-end. Generating a proof does thus not depend on the amount of states visited, and the time needed for writing the proof is negligible in consideration of the time $h^C$ spends on exploring and refining its heuristic.

Regarding memory usage, we see that $h^2$ and $h^{max}$ can cause significant memory overhead, while $h^{M\&S}$ and $max(h^{M\&S}, h^2)$ sometimes cause moderate overhead and $h^C$ none. For all configurations, there are two major potential origins of overhead: (1) we need to store a BDD of all expanded states, and (2) heuristics might need to store information about dead-ends in order to write their sub-proof later on.

The significant memory use of $h^2$ and $h^{max}$ can be explained with the latter cause. We assume that $max(h^{M\&S}, h^2)$ is not as affected by this since $h^{M\&S}$ is evaluated before $h^2$ and might thus detect the majority of dead-ends before $h^2$ is involved. The increased memory requirement for $h^{M\&S}$ on the other hand is most likely due to a large set of expanded states, which also explains why these cases rather occur in large tasks. Finally, $h^C$ does not need to store expanded state nor heuristic information, thus incurring no memory overhead.

| | $h^{\max}$ | | | $h^{\text{M\&S}}$ | | |
|---|---|---|---|---|---|---|
| | finished/#proofs | memory | time | finished/#proofs | memory | time |
| $\text{Ver}^{\text{P}}$ | 156/172 | 0 | -16 | 198/212 | -14 | 0 |
| $\text{Ver}^{\text{P}'}$ | 162/168 | -1 | -5 | 198/212 | -14 | 0 |

Table 10.2: Reasons for failure of the verifier. "finished" denotes the number of tasks where the proof could be verified within the given limits.

## 10.3 Verification

Verifying the generated proofs succeeded in all cases for $h^2$, $max(h^{\text{M\&S}}, h^2)$ and all but one case for $h^C$ which failed due to a timeout. Of the proofs generated by $h^{\text{M\&S}}$, 14 could not be verified, all of them due to a reached memory limit; and for $h^{\max}$ 16 proof verifications failed, this time solely due to a timeout. The memory failures in $h^{\text{M\&S}}$ are related to the amount of detected dead-ends in the corresponding task: all such proofs contained over 3,500,000 dead-ends. The reason this causes high memory usage is that proof contains one MODS formula for each dead-end, one MODS formula containing *all* dead-ends (whose size is linear in the amount of dead-ends), one BDD formula containing *all* dead-ends and a number of formulas (for $h^{\text{M\&S}}$ this is one BDD) needed in the sub-proofs for each dead-end.

In order to reduce memory usage for the verifier, we implemented a variant $\text{Ver}^{\text{P}'}$ in which the verifier discards formulas once they are not needed anymore. For this, the verifier first reads through the entire proof and remembers for each formula which judgment is the last that needs it. This also necessitated a change when generating the proof: similar to $\text{FD}^{\text{C}'}$, $\text{FD}^{\text{P}'}$must now write BDDs individually, causing more time overhead and increased certificate size.

We tested $\text{FD}^{\text{P}'}$and $\text{Ver}^{\text{P}'}$with $h^{\max}$ and $h^{\text{M\&S}}$. Table 10.2 compares $\text{Ver}^{\text{P}'}$and $\text{Ver}^{\text{P}}$. We see that discarding formulas could not avoid running out of memory in those proofs that could not be verified by $\text{FD}^{\text{P}}$. However, we are able to verify more proofs for $h^{\max}$, even though we loose coverage in generating proofs. We assume that discarding the formulas saves enough memory such that CUDD does not need to slow down in order to release memory.[1]

Figure 10.2 shows the time needed for verification as a function of the proof size. Overall the relation between these two metrics appears to be linear, but with clear differences between the configurations. $h^{\text{M\&S}}$ proofs are in general the fastest to verify relative to their certificate size, although several outliers exist especially for small certificates. We assume this is because $h^{\text{M\&S}}$ shares the sub-proof for each dead-end in one BDD. Configurations containing a form of $h^C$(i.e. $h^2$, $max(h^{\text{M\&S}}, h^2)$ and $h^C$) on the other hand tend to have very compact proofs which are harder to verify. This trend seems to be related to the complexity of evaluating $h^C$ itself: as we consider more total tuples in $h^C$, more tuples are unreachable and thus the Horn formula denoting the inductive set grows. Furthermore, if

[1] Table B.3 in Appendix B shows full coverage results for $\text{FD}^{\text{P}'}$.

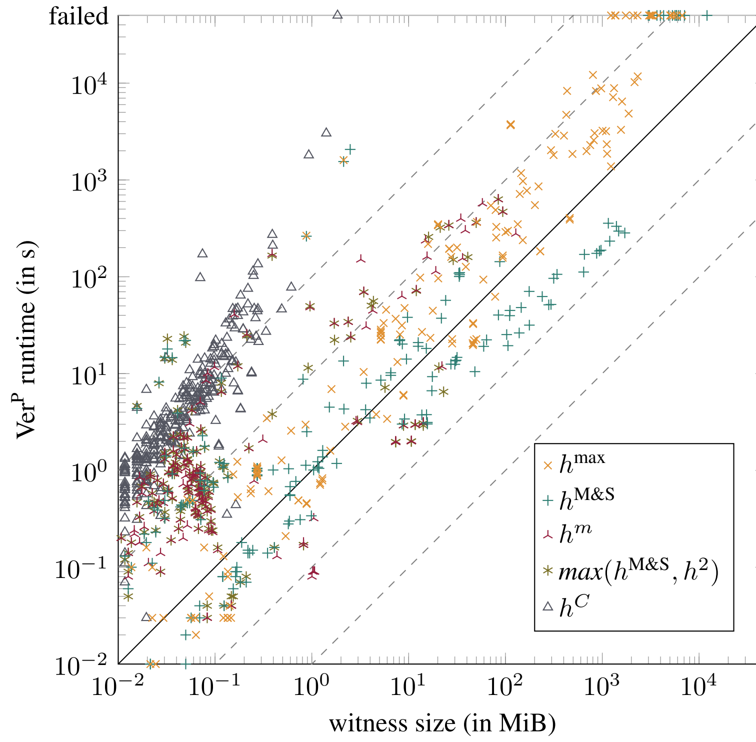



Figure 10.2: Time needed for verification of the proof as a function of its size. Only tasks for which a proof could be generated are shown.

we have more variables in a tuple the corresponding clause is longer. $max(h^{\text{M\&S}}, h^2)$ does not suffer as much from this as $h^2$ does since $h^{\text{M\&S}}$ likely covers many dead-ends already. $h^C$ on the other hand is even more expensive to verify since it most likely contains more tuples than $h^2$ and a fair amount of those tuples contain more than two variables.

# 11 Comparison

After analyzing both inductive certificates and our proof system individually, this chapter compares their performance on configurations $h^{\max}$ and $h^{\text{M\&S}}$ with the goal to identify cases where one witness type is preferred over the other one. Our comparison will investigate three factors: the size of the generated witness, the overhead incurred on the planning system and the performance of the verifier.

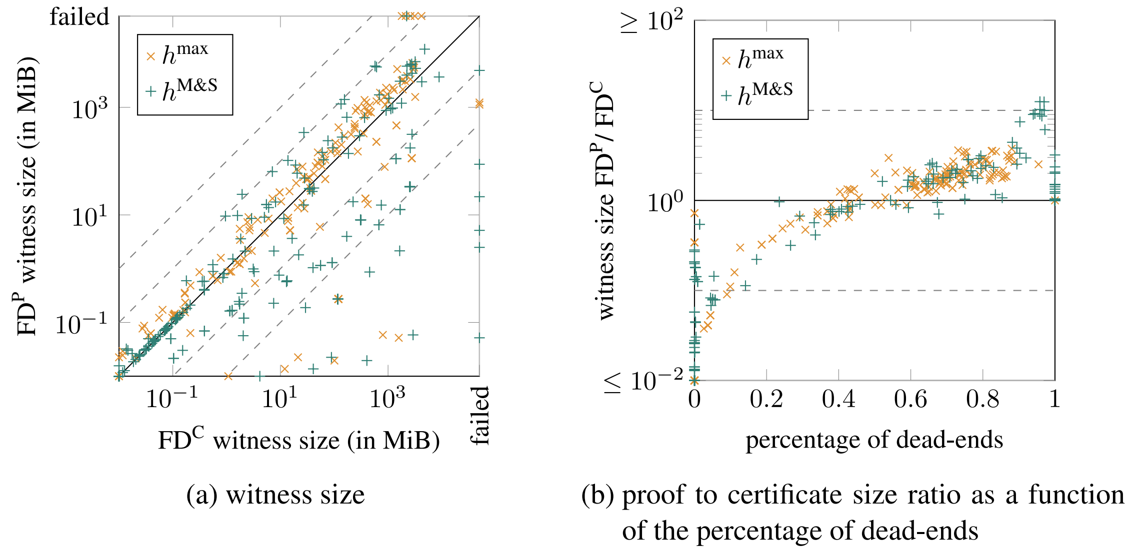
## 11.1 Witness Size

Comparing the size of the generated witnesses as shown in Figure 11.1 (a) yields mixed results. For $h^{\max}$, inductive certificates are often smaller than proofs, although extremes in the other direction exist. The size variations for $h^{\text{M\&S}}$ are more diverse, but overall proof sizes tend to be smaller — often quite significantly.

In general, proofs need to output more information for each dead-end. While certificates only need to print the set itself, proofs also need to print a sub-proof as to why this dead-end is dead; and need to build the abstract union of all dead-ends and show it is dead one step at a time. On the other hand, certificates need to output more information for each expanded state, specifically the BDD for this state as well as for each successor a hint which set holds this successor. In comparison, proofs only need to output one BDD containing all expanded states, as well as a short proof why this set is dead. It thus stands to reason that proofs are smaller than certificates if the majority of visited states has been expanded, and bigger if the majority are dead-ends. Indeed, Figure 11.1 (b) supports this analysis. It shows the factor by which the proof is bigger than the certificate as a function of the percentage of dead-ends in all visited states. As the percentage of dead-end increases, proofs become bigger than certificates.

## 11.2 Generation

In order to produce a witness, $\text{FD}^{\text{C}}$ and $\text{FD}^{\text{P}}$ need to do similar things: they need to collect all expanded states, all dead-ends and a number of inductive sets covering all dead-ends. Thus there are not many significant differences in the runtime of the certifying planner, as we see in Figure 11.2. Overall we can see however that $\text{FD}^{\text{P}}$ seems to cause more overhead to the planner than $\text{FD}^{\text{C}}$. For $h^{\max}$ this increase in overhead seems to be proportional to the runtime, while for $h^{\text{M\&S}}$ there is more variation, especially for longer runtimes. Table 11.1 provides a different view on this data by showing for how many tasks one




(a) witness size

(b) proof to certificate size ratio as a function of the percentage of dead-ends

Figure 11.1: Comparison of the witness size between FD$^C$ and FD$^P$. Tasks for which neither method could create a witness are omitted, for (b) only tasks where both created a witness are considered.

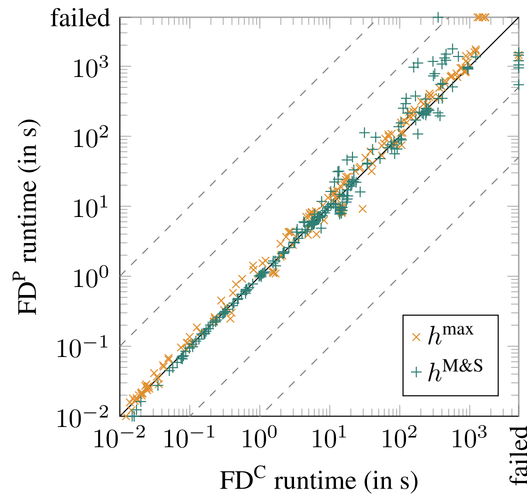



Figure 11.2: Time usage comparison between generating certificates and proofs. Tasks for which neither method could create a witness are omitted.

| | FD$^C$ | | FD$^P$ | |
|---|---|---|---|---|
| | + | faster | faster | + |
| $h^{max}$ | 5 | 126 | 44 | 2 |
| $h^{M\&S}$ | 1 | 85 | 121 | 6 |
| total | 6 | 211 | 165 | 8 |

Table 11.1: Time comparison for generating witnesses between FD$^C$ and FD$^P$. "+" means that the corresponding certifying algorithm succeeded in generating a witness while the other did not.

| | FD$^C$ | | FD$^P$ | |
|---|---|---|---|---|
| | + | faster | faster | + |
| $h^{max}$ | 0 | 26 | 120 | 8 |
| $h^{M\&S}$ | 1 | 54 | 132 | 7 |
| total | 1 | 80 | 252 | 15 |

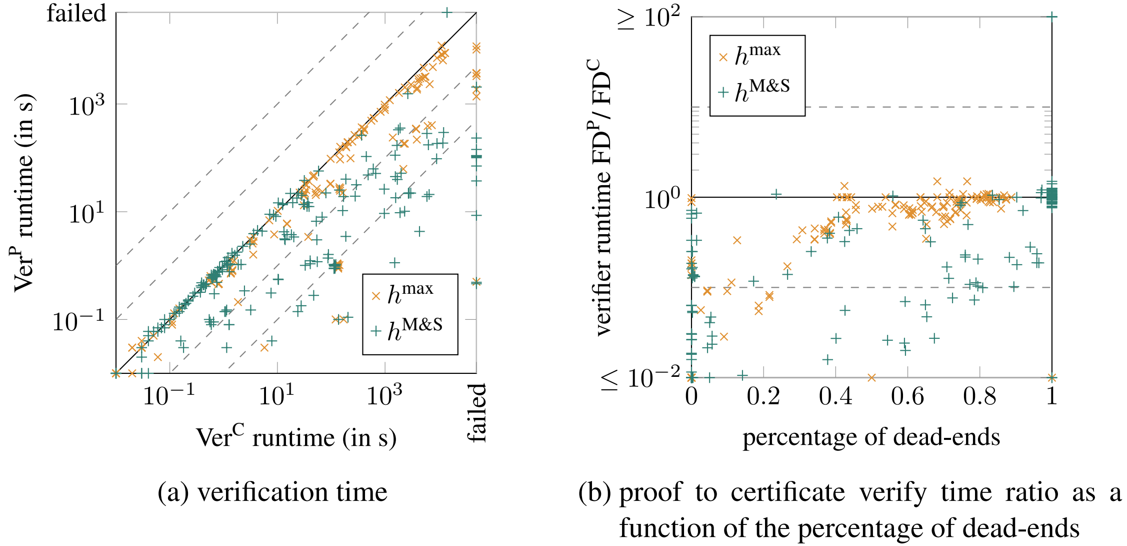
Table 11.2: Time comparison for verifying witnesses between FD$^C$ and FD$^P$. "+" means that the corresponding certifying algorithm succeeded in verifying a witness while the other did not. Only tasks where witnesses of both types were generated are considered.

certifying algorithm could create a witness where the other one could not, or was faster in case both succeeded.

One reason why FD$^P$ requires more time generating the proof relates to the amount of dead-ends. As discussed in the previous section, proofs tend to become bigger as the ratio of dead-ends increased, and as such FD$^P$ requires more time to write the witness for tasks with many dead-ends. Following this line of thought we would assume that FD$^P$ would be significantly faster in cases where the proof is substantially smaller than the certificate, but this is not the case. The size reduction is a result of only building one BDD containing *all* expanded states. But building this BDD requires to first build a BDD of each expanded state (which FD$^C$ does as well), and then additionally build the intersection of these BDDs (which FD$^C$ does not do). Thus, while expanded states are represented more compactly in proofs, FD$^P$ needed more time to obtain this representation.

## 11.3 Verification

While FD$^C$ and FD$^P$ are comparable when generating a witness, verifying the witness yields vastly different results. As shown in Table 11.2 FD$^P$ could verify all but one task FD$^C$ could (the one task nearly reaching a timeout in FD$^C$) as well as 15 tasks where FD$^C$ failed; and was faster in most cases where both could verify their witness. Figure 11.3 (a) further emphasizes these results, showing that verification with FD$^P$ is frequently faster by a factor of 10, and in extreme cases by a factor of 1000.




(a) verification time

(b) proof to certificate verify time ratio as a function of the percentage of dead-ends

Figure 11.3: Comparison of verification time between Ver$^C$ (with certificates generated by FD$^C$) and Ver$^P$. Tasks for which neither method could create a witness are omitted, for (b) only tasks where both succeeded in creating and verifying a witness are considered.

The reason behind these results lies in the difference how expanded states are treated. The 1-disjunctive certificate generated by FD$^C$ in its essence rebuilds an explicit search, considering each expanded state individually. The proof generated by FD$^P$ on the other hand requires a bit more work to establish the set of all dead-ends as dead, but then can conclude the proofs within a few steps by looking at the progression of *all* expanded states.

Figure 11.3 (b) shows that, as with witness size, FD$^C$ tends to do better when the percentage of dead-ends is higher, and can even be faster than FD$^P$ in these cases. But unlike the witness size it never shows a considerable advantage against FD$^P$.

## 11.4 Summary

Our experimental evaluation demonstrated that both inductive certificates and proofs in our proof system are a viable witness for heuristic search. While inductive certificates have some advantages in causing less overhead when generating witnesses, they suffer from two major disadvantages:

- They cannot be used for heuristic search with multiple heuristics requiring different representation formalisms.
- They rebuild an explicit search and are thus slow to verify.

The proof system on the other hand is capable of combining information from a variety of sources, as long as those sources can generate a sub-proof of some form; and can form more concise overall proofs since sub-proofs are only loosely coupled in the sense that the exact reasoning does not matter once the judgment has been verified.

We believe that inductive certificates can be useful in cases where the reasoning of the algorithm very closely fits the idea of inductive certificates. In these cases this reasoning is already inherent in the certificates, while the proof system (while capable of generating a proof) must express it explicitly, leading to higher overhead. For a more general usage however, the proof system offers more versatility and a richer expressiveness, and is thus the preferred choice.

# Part IV

# Future Work and Conclusion

# 12 Future Work

We believe that both inductive certificates and proofs in our proof system have great potential to serve as witness for a wide variety of planning techniques not discussed in this thesis, possibly by extending their definition further. In what follows, we discuss several ideas on both how to extend the witness definition and how new techniques could make use of them.

## 12.1 Witnesses

**Inductive Certificates** The idea of composite certificates could be developed further by allowing more general compositions, for example a union of intersections or even an arbitrary composition. While it will not be possible to allow arbitrary compositions and guarantee efficient verification, we could possibly define similar restrictions as in $r$-conjunctive and $r$-disjunctive certificates. For example one could define a certificate akin to an $r$-disjunctive certificate, but each member is represented by a conjunction of sets and to show inductivity it suffices to consider only up to $r'$ members of the conjunction. This would allow us to use inductive certificates in cases where it is not feasible to represent the inductive set as a simple conjunction or disjunction.

Another possible extension is to consider backwards-inductive certificates and investigate how they relate to forward-inductive certificates. For example, instead of a simple inductive certificate $S$, we could define a backwards inductive certificate $\overline{S}$ which does not contain the initial state, contains all goal states, and is backwards inductive. These types of certificates could be particularly useful for formalisms not supporting $\neg$**C** if we cannot express an inductive certificate $S$ but can express its backwards counterpart $\overline{S}$.

**Proof System** The proof system is easily extensible both in its inference rules and basic statements. Whenever a new technique cannot generate a proof in the current definition of the proof system, we can investigate whether adding new inference rules or basic statements enables us to cover this technique. New inference rules could also make use of the assumption mechanism described in Chapter 5.1, which we did not need for the current rules. We need to keep in mind however that one strength of a proof system is having a small core, making its correctness easier to prove. It is thus advisable to be reserved about extending the system in order to avoid adding unnecessary complexity.

One area where extending the system could be beneficial is set theory: more rules from set theory enable a richer possibility of reasoning with sets. For example, in the proof of

Theorems 5.10 and 5.11 we needed two steps to derive $S[A^{\Pi}] \sqsubseteq (S \cup \emptyset)$ from $S[A^{\Pi}] \sqsubseteq S$ because the proof system does not recognize $\emptyset$ as the neutral element of set union. Or consider the alternative proof for Trapper, which cannot be expressed in the current proof system due to a lack of appropriate set theory rules.

Finally, instead of adding new inference rules it might be beneficial to introduce a form of macro rules, i.e. rules that can be deduced within the proof system. For example, introducing a rule stating a set is always a subset of a union containing it would shorten the proof translation of $r$-disjunctive and $r$-conjunctive certificates significantly.

## 12.2 Applications

**Search Algorithms** In this thesis we focused on heuristic explicit forward search since it is one of the most commonly applied planning techniques. There are however many other search algorithms for which we could potentially provide a witness in case of unsolvability, and we present here two examples where we believe that creating witnesses is possible with ideas similar to the ones presented in this thesis.

While we briefly discussed blind symbolic search, state-of-the-art planning systems based on symbolic search like SymBA* (Torralba et al., 2014) employ more advanced strategies, such as using heuristics to guide the search or pruning based on forward and backward mutexes. We believe it to be possible that these planners can emit witnesses in a similar fashion as their explicit counterpart.

A third commonly used planning technique is *planning as satisfiability*, where the planning task is encoded into a propositional formula. Traditional algorithms encode the task only over a finite horizon, i.e. consider only up to $n$ action applications, since each action application increases the formula size. As such, they are unable to prove unsolvability. Recent algorithms based on *property directed reachability* (Suda, 2014) however can avoid the finite horizon problem and can detect unsolvability with an argument similar to inductive sets.

**Heuristics** For heuristics based on delete-relaxation it might be worth investigating whether we can use a single formula representing all dead-ends of the delete-relaxed task. Such an encoding is possible (Muise, 2018) but requires auxiliary variables, a case we did not consider so far. Such formulas could be integrated by using the forget-operator to eliminate the auxiliary variables. Since forget is usually not efficiently supported, it might also be beneficial to investigate if we could avoid it with the help of Lemma 4.1.

For *non-linear* merge strategies, Helmert, Röger, and Sievers (2015) have shown that building an ADD that represents the heuristic can lead to super-polynomial blowup. A recent Master's thesis in our research group (Locher, 2019) suggests that *Sentential Decision Diagrams* (SDD), a generalization of BDDs, cannot avoid the super-polynomial blowup but a formal proof is still pending.

Finally, while we covered several commonly used heuristics, many more have yet to be investigated. One way to potentially cover them is by considering compilations between heuristics as shown in Helmert and Domshlak (2009). For others, new ideas might be necessary. For example, dead-end potential heuristics (Seipp et al., 2016) detect tasks unsolvable by showing that a function over states exists that is nonincreasing (i.e. for each state $s$, all successors $s'$ have a lower or equal function value), but the initial state has a lower function value than any goal state. Such a function in fact defines a family of inductive certificates, more specifically one for each possible function value. Given a value $x$, the set of all states with a function value lower or equal to $x$ is inductive, since it is not possible to reach a state with higher function value than the state itself has. How this inductive set can be represented as a formula is however still an open question.

**Other Approaches** As with heuristics, a variety of planning techniques has yet to be investigated. One direction for possible further research we want to highlight here is a pruning technique called *partial order reduction*. It was first introduced in the area of model checking (e.g. Valmari, 1989) and has also been used in planning (Alkhazraji et al., 2012; Wehrle and Helmert, 2012, 2014).

In a nutshell partial order reduction tries to reduce the search space by avoiding permutation of action sequences yielding the same result. For example, if we need to turn on $n$ light switches, it does not matter in which order we do this. But if we were to perform a breath-first search we would explore $2^n$ states before finding a goal. A partial order reduction could now tell us for example that in the initial state we only need to consider the action turning the first light switch on and we would still be able to find an optimal plan.

A problematic aspect in translating the arguments of partial order reduction into inductive certificates or proofs is that there is no clear way to derive an inductive set from the pruned states. For heuristics, a dead-end usually has some property that all states reachable from it share, and we exploit this property to define an inductive set. States pruned by partial order reduction on the other hand do not have this.

If we want to cover this technique with the current proof system, we need to be able to somehow compactly express all states that were pruned. One possible approach is to try and express this set through a *regression*. A preliminary investigation suggests that all states pruned by a single application of a partial order reduction technique called *strong stubborn sets* to state $s$ can be reached in a single regression step yielding some set $S$. However, $S$ is an *overapproximation* of the desired state set, i.e. it contains states that were not reachable from the pruned state. If we could eliminate these states, then the progression of $S$ would only lead to itself and to expanded states. How to eliminate the unreachable states is as of yet still an open question, as well as how to generalize this observation when partial order reduction is applied multiple times. Alternatively we could try to introduce new rule types which express the conditions for pruning more directly, for example rules denoting that an action can be considered dead under certain circumstances (such as “dead for certain states”).

On a final note, we could also consider to investigate correctness guarantees for the verifiers. While the verifiers are far less complex than planning systems, they can still contain bugs; and a faulty verifier jeopardizes the entire concept of a certifying algorithm. To eliminate this possibility, we could *verify the verifier* with the help of automated theorem provers.

# 13 Conclusion

The aim of this thesis was to provide suitable witnesses for unsolvable planning tasks, where "suitable" is measured based on whether the witness is sound and complete, efficient to generate, efficient to verify and whether it can be applied to a wide variety of state-of-the-art planning techniques.

In Part I, we introduced two different types of witnesses, inductive certificates and proofs in a proof system specifically tailored for proving unsolvability. Both types are built on sets of states represented as a propositional formula in some formalism such as BDDs, Horn formulas, 2CNF formulas or MODS.

Inductive certificates are based on the idea of finding an invariant that holds in the initial state but not in any goal state. They are sound since such an invariant directly implies unsolvability, and complete since for any unsolvable planning task the set of states reachable from the initial state is an inductive certificate. We have shown they can be efficiently verified if the formalism used to represent them efficiently supports certain operations, which all four formalisms mentioned above do. Since an inductive certificate must be represented as a single set of states but oftentimes we cannot compactly represent such a set, we further introduced $r$-disjunctive and $r$-conjunctive certificates. They enable us to define the set as a union or intersection of states, and can be efficiently verified without needing to explicitly build the full disjunction or conjunction.

Our proof system offers a more versatile approach where we can incrementally show that parts of the search space are dead until we have shown the initial state or all goal states to be dead. As with inductive certificates, we have shown that witnesses in the proof system are sound and complete, and can be verified efficiently given the used formalisms efficiently support the required operations. Unlike inductive certificate the proof system allows a mixture of formalisms to be used and enables us to combine information from various sources requiring different formalisms.

A theoretical comparison between the two witness types yielded that all three variations of inductive certificates can be translated into a proof in the proof system, although for $r$-disjunctive and $r$-conjunctive this translation can cause a non-negligible increase in size since reasoning inherent in inductive certificate has to be made explicit in the proof.

In Part II, we exhibit the generality of both approaches by showing how they can be efficiently generated for a multitude of planning techniques. For blind search algorithms, witnesses can be produced fairly easily since the set of expanded states (or its complement for regression) forms an inductive certificate. Our main focus was on explicit heuristic forward search, where heuristics might prune states that they recognize cannot lead to a goal. We have shown however that we can still efficiently generate proofs and to a lim-

ited extent inductive certificates as long as the heuristics involved can produce a reason within the witness for why the state can be pruned. We then covered for several prominent heuristic families how they can do so. Finally we have demonstrated that our witnesses are applicable beyond pure heuristic search by providing instructions on how we can generate witnesses for three planning techniques not fitting into the normal heuristic search schema.

To back up our theoretical claims, we empirically evaluated both generation and verification of the presented witnesses in Part III. To this end, we augmented the Fast Downward planning system to generate inductive certificates and proofs for $A^*$ search with several heuristics, as well as the implementation of clause learning state space search to produce proofs for the setting where the initial state is eventually detected as a dead-end. For verifying the produced witnesses, we implemented two verifiers, one for each witness type. The verifier for inductive certificates is capable of verifying all three types of inductive certificates given as BDDs. The verifier for proofs can handle all four formalisms we considered.

Our experimental study confirmed that producing and verifying witnesses is not only theoretically possible but also practically feasible. Given the same time and memory limits as the original planning algorithm, our certifying version could produce witnesses of both types for a large majority of tasks. From these witnesses, again a large majority could also be verified with the same memory and a more generous time limit.

Comparing the empirical performance between inductive certificates and proofs, proofs came out as clear winners, verifying its witness sometimes several orders of magnitude faster. Inductive certificates however incur less overhead for the certifying planner during witness generation, and also perform strongly in tasks that have a high percentage of dead-ends.

In summary, both witnesses presented here are suitable under the measurements of soundness, completeness, efficiency and generality, and are a first step towards fully certifying planning systems. We do believe however that they can be further improved in order to cover more techniques: inductive certificates could allow more compositions than just a simple union or intersections, and the proof system could introduce new rules, both in set theory and providing new ways of proving state sets dead. But even in their current state we believe that several planning techniques not discussed in this thesis can be covered, such as heuristics like dead-end potentials, planning algorithms like planning as satisfiability or other techniques like partial order reduction.

# Appendix

# A Proof Details

## A.1 Complexity of BDD Operations

This section shows that some BDD operations have tighter bounds when the involved BDDs have specific properties. We assume a BDD implementation as described in Brace, Rudell, and Bryant (1990), which is also used in CUDD (Somenzi, 2015). In this implementation, a BDD encoding of formula $\varphi$ is represented by a node $N$, and $v(N)$ is the lowest ordered variable in $\varphi$ (i.e. $v(N) \preceq v$ for all $v$ occurring in $\varphi$). We define $N_v$ as the BDD representing $\varphi_v$ which restricts $v$ in $\varphi$ to $\top$, and $N_{\neg v}$ as the BDD representing $\varphi_{\neg v}$ which restricts $v$ in $\varphi$ to $\bot$ (if $v$ does not occur in the formula represented by $N$ we have $N_v = N_{\neg v} = N$). $N$ has two outgoing edges: the if-edge leads to $N_{v(N)}$ and the else-edge to $N_{\neg v(N)}$. Constant function $\top$ and $\bot$ have no outgoing edges and no associated variable.

Most operations on BDDs are done by calling a function $\mathtt{ite}(X, Y, Z)$ which stands for "if $X$ then $Y$ else $Z$". `ite` works in a recursive fashion: it first determines the lowest ordered variable $v_l$ among $v(X)$, $v(Y)$, and $v(Z)$; and as a result returns a node $N$ with if-edge $\mathtt{ite}(X_{v_l}, Y_{v_l}, Z_{v_l})$, else-edge $\mathtt{ite}(X_{\neg v_l}, Y_{\neg v_l}, Z_{\neg v_l})$ and $v(N) = v_l$. The terminal cases of the recursion are:

- ite$(\top, N, N') = N$,
- ite$(\bot, N, N') = N'$ and
- ite$(N, \top, \bot) = N$.

Furthermore, if the if- and else-edge are the same node $N$ then $N$ is returned, and nodes are stored in a unique table which allows to only generate new nodes if an isomorphic node does not exist yet (these two enhancements guarantee the result to already be reduced). We assume everything other than the recursion is done in constant time.

**Theorem A.1.** *Building the union of two BDDs where one BDD represents a complete variable assignment (or "adding a model to a BDD") is possible in time linear in the amount of variables $V$. More concretely, the* `ite` *recursion is only called $2 \cdot |V| + 1$ times and at most $|V|$ new nodes are created.*

**Proof:** The union of two BDDs $X$ and $Y$ is built by calling $\mathtt{ite}(X, \top, Y)$. We first assume $X$ is a complete assignment, meaning that $v(X)$ is the lowest ordered variable and either $X_{v(X)}$ or $X_{\neg v(X)}$ is equal to $\bot$, while the other one is a BDD representing an assignment to all variables other than $v(X)$ (or to all $v \succ v(X)$. We will show

the claim with induction over the size of $V$. If $V = \{v\}$, then $\texttt{ite}(X, \top, Y)$ will call $\texttt{ite}(X_v, \top_v(= \top), Y_{v_0})$ and $\texttt{ite}(X_{\neg v}, \top, Y_{\neg v})$ (since $v$ is the lowest ordered variable and occurs in X). Since $X$ and $Y$ depend only on $v$ (or $Y$ potentially on nothing), restricting them on $v$ will lead to constant functions, meaning both ite calls will be terminal, leading to a total of $3(= 2 \cdot |V| + 1)$ calls, and only the top call potentially creates a new node. For $V = \{v_0 \prec \cdots \prec v_n\}$, we have on the top level recursion the two calls $\texttt{ite}(X_{v_0}, \top, Y_{v_0})$ and $\texttt{ite}(X_{\neg v_0}, \top, Y_{\neg v_0})$ (since $v_0$ is the lowest ordered variable and occurs in $X$). We know that either $X_{v_0}$ or $X_{\neg v_0}$ must be equal to $\bot$, meaning one of the calls has $\bot$ as first argument and is thus terminal. The arguments of the other call can be seen as a union of two formulas over variable set $V' = V \setminus \{v_0\}$, where $X$ is a complete variable assignment, meaning it will require $2 \cdot |V'| + 1$ ite calls and generate at most $|V'|$ new nodes according to our induction assumption. Together with the top level `ite` call, the terminal call and possibly a new node in the top level `ite` call we have $2 \cdot |V'| + 1 + 2 = 2 \cdot |V| + 1$ calls and at most $|V'| + 1 = |V|$ new nodes. All proof steps for the case of $Y$ being the complete assignment are analogous, except that the terminal call is terminal because the second argument is $\top$ and the third $\bot$, rather than because the first argument is $\bot$. $\square$

**Theorem A.2.** *Let $X$ and $Y$ be two BDDs such that $v(X) \prec v(Y)$. Given some $v_i \succ v(X)$ let $V' = \{v' | v(X) \preceq v' \preceq v_i\}$, and $X_{V'}$/$Y_{V'}$ be X/Y restricted to $V'$ (i.e. forgetting all $v \notin V'$). If*

- *$X_{V'}$ represents a complete assignment to variables $v' \in V'$, and*
- *$X_{V'}$ and $Y_{V'}$ are disjoint (i.e. $X_{V'} \wedge Y_{V'} \equiv \bot$),*

*then building $X \vee Y$ is linear in $|V'|$.*

**Proof:** Since $v(X) \prec v(Y)$, we can treat $v(X)$ as lowest ordered variable. We have a union where on one side the first $|V'|$ variables are uniquely assigned, thus up to this step we have the same recursion behavior as in Theorem A.1. At variable $v_i$ we only have one node to generate with if-child $\texttt{ite}(X_{v_i}, \top, Y_{v_i})$ and else-child $\texttt{ite}(X_{\neg v_i}, \top, Y_{\neg v_i})$. Since we know that after this variable $X$ and $Y$ become disjoint either $X_{v_i}$ or $Y_{v_i}$ must be equal to $\bot$, and analogously for $X_{\neg v_i}$ and $Y_{\neg v_i}$. Either way both calls must thus be terminal, leading to a total of $2 \cdot |V'| + 1$ calls and at most $|V'|$ new nodes. $\square$

**Theorem A.3.** *Building the intersection of two BDDs $X$ and $Y$ where all variables occurring in $X$ are ordered lower than any variable occurring in $Y$ is linear in the representation size of $X$.*

**Proof:** The intersection of two BDDs $X$ and $Y$ is built by calling $\texttt{ite}(X, Y, \bot)$. The recursion goes through each node connected to $X$, always calling $\texttt{ite}(X'_v, Y_v = Y, \bot)$ and $\texttt{ite}(X'_{\neg v}, Y_v = Y, \bot)$ for some restricted form of $X$. Each call where $X'_v$ or $X'_{\neg v}$ leads to $\top$ or $\bot$ is terminal and has $Y$ or $\bot$ as child; thus we need to create at most $\|X\|$ new nodes. $\square$

## A.2 Translation of Inductive Certificates to Proofs

We present full schematic translations for $r$-disjunctive (Table A.1) and $r$-conjunctive certificates (Table A.2) to proofs in the proof system. In what follows, $x - i$ denotes the judgment derived i steps previously in a *concrete* proof, not in the schematic proof.

Tables A.3 and A.4 exemplify the schemata by translating the certificates from Figure 4.4 and 4.5.

**$r$-disjunctive certificate schema:**

| # | judgment | rule | premises |
|---|---|---|---|
| (1) | $A^{\Pi} \sqsubseteq (\dots(a_1 \cup a_2) \cdots \cup a_m)$ | **B5** | |
| (2) | $\emptyset$ dead | **ED** | |
| (3) | $S_1 \sqsubseteq (S_1 \cup S_2)$ | **UR** | |
| (4) | $(S_1 \cup S_2) \sqsubseteq ((S_1 \cup S_2) \cup S_3)$ | **UR** | |
| ... | | | |
| (5) | $(\dots(S_1 \cup S_2) \cdots \cup S_{n-1}) \sqsubseteq \mathcal{F}_{\cup}$ | **UR** | |
| (6) | $S_1 \sqsubseteq ((S_1 \cup S_2) \cup S_3)$ | **ST** | (3),(4) |
| ... | | | |
| (7) | $S_1 \sqsubseteq \mathcal{F}_{\cup}$ | **ST** | $(7-1)$,(5) |
| (8) | $S_2 \sqsubseteq (S_1 \cup S_2)$ | **UL** | |
| ... | | | |
| (9) | $S_2 \sqsubseteq \mathcal{F}_{\cup}$ | **ST** | $(9-1)$,(5) |
| ... | | | |
| (10) | $S_n \sqsubseteq \mathcal{F}_{\cup}$ | **UL** | |
| ... | | | |
| (11) | $(S_1^{1,1} \cup S_2^{1,1}) \sqsubseteq \mathcal{F}_{\cup}$ | **SU** | $(i_1^{1,1})(i_2^{1,1})$ |
| ... | | | |
| (12) | $\mathcal{F}_{\cup}^{1,1} \sqsubseteq \mathcal{F}_{\cup}$ | **SU** | $(12-1)$,$(i_k^{1,1})$ |
| (13) | $S_1[a_1] \sqsubseteq \mathcal{F}_{\cup}^{1,1}$ | **B2** | |
| (14) | $S_1[a_1] \sqsubseteq \mathcal{F}_{\cup}$ | **ST** | (13),(12) |
| ... | | | |
| (15) | $S_1[a_2] \sqsubseteq \mathcal{F}_{\cup}$ | **ST** | $(15-1)$,$(15-2)$ |
| ... | | | |
| (16) | $S_1[a_m] \sqsubseteq \mathcal{F}_{\cup}$ | **ST** | $(16-1)$,$(16-2)$ |
| (17) | $S_1[(a_1 \cup a_2)] \sqsubseteq \mathcal{F}_{\cup}$ | **AU** | (14),(15) |
| ... | | | |
| (18) | $S_1[(\dots(a_1 \cup a_2) \cdots \cup a_m)] \sqsubseteq \mathcal{F}_{\cup}$ | **AU** | $(18-1)$,(16) |
| (19) | $S_1[A^{\Pi}] \sqsubseteq \mathcal{F}_{\cup}$ | **AT** | (18),(1) |
| ... | | | |
| (20) | $S_2[A^{\Pi}] \sqsubseteq \mathcal{F}_{\cup}$ | **AT** | $(20-1)$,(1) |
| ... | | | |

| # | judgment | rule | premises |
|---|---|---|---|
| (21) | $S_n[A^\Pi] \sqsubseteq \mathcal{F}_\cup$ | **AT** | (21 − 1),(1) |
| (22) | $(S_1 \cup S_2)[A^\Pi] \sqsubseteq \mathcal{F}_\cup$ | **PU** | (19),(20) |
| … | | | |
| (23) | $\mathcal{F}_\cup[A^\Pi] \sqsubseteq \mathcal{F}_\cup$ | **PU** | (23 − 1),(21) |
| (24) | $\mathcal{F}_\cup \sqsubseteq (\mathcal{F}_\cup \cup \emptyset)$ | **UR** | |
| (25) | $\mathcal{F}_\cup[A^\Pi] \sqsubseteq (\mathcal{F}_\cup \cup \emptyset)$ | **ST** | (23),(24) |
| (26) | $(S_1 \cap S_G^\Pi) \sqsubseteq \emptyset$ | **B1** | |
| (27) | $(S_2 \cap S_G^\Pi) \sqsubseteq \emptyset$ | **B1** | |
| (28) | $((S_1 \cap S_G^\Pi) \cup (S_2 \cap S_G^\Pi)) \sqsubseteq \emptyset$ | **SU** | (26),(27) |
| (29) | $((S_1 \cup S_2) \cap S_G^\Pi) \sqsubseteq ((S_1 \cap S_G^\Pi) \cup (S_2 \cap S_G^\Pi))$ | **DI** | |
| (30) | $((S_1 \cup S_2) \cap S_G^\Pi) \sqsubseteq \emptyset$ | **ST** | (29),(28) |
| … | | | |
| (31) | $(\mathcal{F}_\cup \cap S_G^\Pi) \sqsubseteq \emptyset$ | **ST** | (31 − 1),(31 − 2) |
| (32) | $(\mathcal{F}_\cup \cap S_G^\Pi)$ dead | **SD** | (2),(31) |
| (33) | $\mathcal{F}_\cup$ dead | **PG** | (24),(2),(32) |
| (34) | $\{I^\Pi\} \sqsubseteq S_i$ | **B1** | |
| (35) | $\{I^\Pi\} \sqsubseteq \mathcal{F}_\cup$ | **ST** | (34),($i_k$) |
| (36) | $\{I^\Pi\}$ dead | **SD** | (33),(35) |
| (37) | unsolvable | **CI** | (36) |

Table A.1: Schematic translation of an $r$-disjunctive certificate to a proof witness.

**$r$-conjunctive certificate schema:**

| # | judgment | rule | premises |
|---|---|---|---|
| (1) | $A^\Pi \sqsubseteq (\ldots(a_1 \cup a_2) \cdots \cup a_m)$ | **B5** | |
| (2) | $\emptyset$ dead | **ED** | |
| (3) | $(S_1 \cap S_2) \sqsubseteq S_1$ | **IR** | |
| (4) | $((S_1 \cap S_2) \cap S_3) \sqsubseteq (S_1 \cap S_2)$ | **IR** | |
| … | | | |
| (5) | $\mathcal{F}_\cap \sqsubseteq (\ldots(S_1 \cap S_2) \cdots \cap S_{n-1})$ | **IR** | |
| (6) | $((S_1 \cap S_2) \cap S_3) \sqsubseteq S_1$ | **ST** | (4),(3) |
| … | | | |
| (7) | $\mathcal{F}_\cap \sqsubseteq S_1$ | **ST** | (5),(7 − 1) |
| (8) | $(S_1 \cap S_2) \sqsubseteq S_2$ | **IL** | |
| … | | | |
| (9) | $\mathcal{F}_\cap \sqsubseteq S_2$ | **ST** | (5),(9 − 1) |
| … | | | |
| (10) | $\mathcal{F}_\cap \sqsubseteq S_n$ | **IL** | |
| (11) | $\mathcal{F}_\cap \sqsubseteq (S_1^{1,1} \cap S_2^{1,1})$ | **SI** | $(i_1^{1,1})(i_2^{1,1})$ |
| … | | | |

| # | judgment | rule | premises |
|---|---|---|---|
| (12) | $\mathcal{F}_\cap \sqsubseteq \mathcal{F}_\cap^{1,1}$ | **SI** | $(12-1)$,$(_k^{1,1})$ |
| (13) | $\mathcal{F}_\cap^{1,1}[a_1] \sqsubseteq S_1$ | **B2** | |
| (14) | $\mathcal{F}_\cap[a_1] \sqsubseteq S_1$ | **PT** | (13),(12) |
| ... | | | |
| (15) | $\mathcal{F}_\cap[a_2] \sqsubseteq S_1$ | **PT** | $(15-1)$,$(15-2)$ |
| ... | | | |
| (16) | $\mathcal{F}_\cap[a_m] \sqsubseteq S_1$ | **PT** | $(16-1)$,$(16-2)$ |
| (17) | $\mathcal{F}_\cap[(a_1 \cup a_2)] \sqsubseteq S_1$ | **AU** | (14),(15) |
| ... | | | |
| (18) | $\mathcal{F}_\cap[(\dots(a_1 \cup a_2) \dots \cup a_m)] \sqsubseteq S_1$ | **AU** | $(18-1)$,(16) |
| (19) | $\mathcal{F}_\cap[A^\Pi] \sqsubseteq S_1$ | **AT** | (18),(1) |
| ... | | | |
| (20) | $\mathcal{F}_\cap[A^\Pi] \sqsubseteq S_2$ | **AT** | $(20-1)$,(1) |
| ... | | | |
| (21) | $\mathcal{F}_\cap[A^\Pi] \sqsubseteq S_n$ | **AT** | $(21-1)$,(1) |
| (22) | $\mathcal{F}_\cap[A^\Pi] \sqsubseteq (S_1 \cap S_2)$ | **SI** | (19),(20) |
| ... | | | |
| (23) | $\mathcal{F}_\cap[A^\Pi] \sqsubseteq \mathcal{F}_\cap$ | **SI** | $(23-1)$,(21) |
| (24) | $\mathcal{F}_\cap \sqsubseteq (\mathcal{F}_\cap \cup \emptyset)$ | **UR** | |
| (25) | $\mathcal{F}_\cap[A^\Pi] \sqsubseteq (\mathcal{F}_\cap \cup \emptyset)$ | **ST** | (23),(24) |
| (26) | $\mathcal{F}_\cap \sqsubseteq (S_1^G \cap S_2^G)$ | **SI** | $(i_1^G)$,$(i_2^G)$ |
| ... | | | |
| (27) | $\mathcal{F}_\cap \sqsubseteq \mathcal{F}_\cap^G$ | **SI** | $(27-1)$,$(i_k^G)$ |
| (28) | $(\mathcal{F}_\cap \cap S_G^\Pi) \sqsubseteq \mathcal{F}_\cap$ | **IR** | |
| (29) | $(\mathcal{F}_\cap \cap S_G^\Pi) \sqsubseteq \mathcal{F}_\cap^G$ | **ST** | (28),(27) |
| (30) | $(\mathcal{F}_\cap \cap S_G^\Pi) \sqsubseteq S_G^\Pi$ | **IL** | |
| (31) | $(\mathcal{F}_\cap \cap S_G^\Pi) \sqsubseteq (\mathcal{F}_\cap^G \cap S_G^\Pi)$ | **SI** | (29),(30) |
| (32) | $(\mathcal{F}_\cap^G \cap S_G^\Pi) \sqsubseteq \emptyset$ | **B1** | |
| (33) | $(\mathcal{F}_\cap \cap S_G^\Pi) \sqsubseteq \emptyset$ | **ST** | (31),(32) |
| (34) | $(\mathcal{F}_\cap \cap S_G^\Pi)$ dead | **SD** | (2),(33) |
| (35) | $\mathcal{F}_\cap$ dead | **PG** | (24),(2),(34) |
| (36) | $\{I^\Pi\} \sqsubseteq S_1$ | **B1** | |
| (37) | $\{I^\Pi\} \sqsubseteq S_2$ | **B1** | |
| (38) | $\{I^\Pi\} \sqsubseteq S_n$ | **B1** | |
| (39) | $\{I^\Pi\} \sqsubseteq (S_1 \cap S_2)$ | **SI** | (36),(37) |
| ... | | | |
| (40) | $\{I^\Pi\} \sqsubseteq F_\cap$ | **SI** | $(40-1)$,(38) |
| (41) | $\{I^\Pi\}$ dead | **SD** | (35),(40) |
| (42) | unsolvable | **CI** | (41) |

Table A.2: Schematic translation of an $r$-conjunctive certificate to a proof witness.

**$r$-disjunctive certificate example:**

| # | judgment | rule | premises |
|---|---|---|---|
| (1) | $A^{\Pi} \sqsubseteq ((a_1 \cup a_2) \cup a_3)$ | **B5** | |
| (2) | $S_1 \sqsubseteq (S_1 \cup S_2)$ | **UR** | |
| (3) | $S_2 \sqsubseteq (S_1 \cup S_2)$ | **UL** | |
| (4) | $\emptyset$ dead | **ED** | |
| (5) | $S_1[a_1] \sqsubseteq S_1$ | **B2** | |
| (6) | $S_1[a_1] \sqsubseteq (S_1 \cup S_2)$ | **ST** | (5),(2) |
| (7) | $S_1[a_2] \sqsubseteq S_2$ | **B2** | |
| (8) | $S_1[a_2] \sqsubseteq (S_1 \cup S_2)$ | **ST** | (7),(3) |
| (9) | $S_1[a_3] \sqsubseteq S_1$ | **B2** | |
| (10) | $S_1[a_3] \sqsubseteq (S_1 \cup S_2)$ | **ST** | (9),(2) |
| (11) | $S_1[(a_1 \cup a_2)] \sqsubseteq (S_1 \cup S_2)$ | **AU** | (6),(8) |
| (12) | $S_1[((a_1 \cup a_2) \cup a_3)] \sqsubseteq (S_1 \cup S_2)$ | **AU** | (11),(10) |
| (13) | $S_1[A^{\Pi}] \sqsubseteq (S_1 \cup S_2)$ | **AT** | (12),(1) |
| (14) | $S_2[a_1] \sqsubseteq S_2$ | **B2** | |
| (15) | $S_2[a_1] \sqsubseteq (S_1 \cup S_2)$ | **ST** | (14),(3) |
| (16) | $S_2[a_2] \sqsubseteq S_2$ | **B2** | |
| (17) | $S_2[a_2] \sqsubseteq (S_1 \cup S_2)$ | **ST** | (16),(3) |
| (18) | $S_2[a_3] \sqsubseteq S_1$ | **B2** | |
| (19) | $S_2[a_3] \sqsubseteq (S_1 \cup S_2)$ | **ST** | (18),(2) |
| (20) | $S_2[(a_1 \cup a_2)] \sqsubseteq (S_1 \cup S_2)$ | **AU** | (15),(17) |
| (21) | $S_2[((a_1 \cup a_2) \cup a_3)] \sqsubseteq (S_1 \cup S_2)$ | **AU** | (20),(19) |
| (22) | $S_2[A^{\Pi}] \sqsubseteq (S_1 \cup S_2)$ | **AT** | (21),(1) |
| (23) | $(S_1 \cup S_2)[A^{\Pi}] \sqsubseteq (S_1 \cup S_2)$ | **PU** | (13),(22) |
| (24) | $(S_1 \cup S_2) \sqsubseteq ((S_1 \cup S_2) \cup \emptyset)$ | **UR** | |
| (25) | $(S_1 \cup S_2)[A^{\Pi}] \sqsubseteq ((S_1 \cup S_2) \cup \emptyset)$ | **ST** | (23),(24) |
| (26) | $(S_1 \cap S_G^{\Pi}) \sqsubseteq \emptyset$ | **B1** | |
| (27) | $(S_2 \cap S_G^{\Pi}) \sqsubseteq \emptyset$ | **B1** | |
| (28) | $((S_1 \cap S_G^{\Pi}) \cup (S_2 \cap S_G^{\Pi})) \sqsubseteq \emptyset$ | **SU** | (26),(27) |
| (29) | $((S_1 \cup S_2) \cap S_G^{\Pi}) \sqsubseteq ((S_1 \cap S_G^{\Pi}) \cup (S_2 \cap S_G^{\Pi}))$ | **DI** | |
| (30) | $((S_1 \cup S_2) \cap S_G^{\Pi}) \sqsubseteq \emptyset$ | **ST** | (29),(28) |
| (31) | $((S_1 \cup S_2) \cap S_G^{\Pi})$ dead | **SD** | (4),(30) |
| (32) | $(S_1 \cup S_2)$ dead | **PG** | (25),(4),(31) |
| (33) | $\{I^{\Pi}\} \sqsubseteq S_1$ | **B1** | |
| (34) | $\{I^{\Pi}\} \sqsubseteq (S_1 \cup S_2)$ | **ST** | (33),(2) |
| (35) | $\{I^{\Pi}\}$ dead | **SD** | (32),(34) |
| (36) | unsolvable | **CI** | (35) |

Table A.3: Translation of the 1-disjunctive certificate from Figure 4.4.

**$r$-conjunctive certificate example:**

| # | judgment | rule | premises |
|---|---|---|---|
| (1) | $A^{\Pi} \sqsubseteq (\ldots(a_1 \cup a_2) \cdots \cup a_m)$ | **B5** | |
| (2) | $\emptyset$ dead | **ED** | |
| (3) | $(S_1 \cap S_2) \sqsubseteq S_1$ | **IR** | |
| (4) | $((S_1 \cap S_2) \cap S_3) \sqsubseteq (S_1 \cap S_2)$ | **IR** | |
| (5) | $((S_1 \cap S_2) \cap S_3) \sqsubseteq S_1$ | **ST** | (4),(3) |
| (6) | $(S_1 \cap S_2) \sqsubseteq S_2$ | **IL** | |
| (7) | $((S_1 \cap S_2) \cap S_3) \sqsubseteq S_2$ | **ST** | (4),(6) |
| (8) | $((S_1 \cap S_2) \cap S_3) \sqsubseteq S_3$ | **IL** | |
| (9) | $S_1[a_1] \sqsubseteq S_1$ | **B2** | |
| (10) | $((S_1 \cap S_2) \cap S_3)[a_1] \sqsubseteq S_1$ | **PT** | (9),(5) |
| (11) | $S_3[a_2] \sqsubseteq S_1$ | **B2** | |
| (12) | $((S_1 \cap S_2) \cap S_3)[a_2] \sqsubseteq S_1$ | **PT** | (11),(8) |
| (13) | $((S_1 \cap S_2) \cap S_3)[(a_1 \cup a_2)] \sqsubseteq S_1$ | **AU** | (10),(12) |
| (14) | $((S_1 \cap S_2) \cap S_3)[A^{\Pi}] \sqsubseteq S_1$ | **AT** | (13),(1) |
| (15) | $((S_1 \cap S_2) \cap S_3) \sqsubseteq (S_2 \cup S_3)$ | **SI** | (7),(8) |
| (16) | $(S_2 \cup S_3)[a_1] \sqsubseteq S_2$ | **B2** | |
| (17) | $((S_1 \cap S_2) \cap S_3)[a_1] \sqsubseteq S_2$ | **PT** | (16),(15) |
| (18) | $S_2[a_2] \sqsubseteq S_2$ | **B2** | |
| (19) | $((S_1 \cap S_2) \cap S_3)[a_2] \sqsubseteq S_2$ | **PT** | (18),(7) |
| (20) | $((S_1 \cap S_2) \cap S_3)[(a_1 \cup a_2)] \sqsubseteq S_2$ | **AU** | (17),(19) |
| (21) | $((S_1 \cap S_2) \cap S_3)[A^{\Pi}] \sqsubseteq S_2$ | **AT** | (19),(1) |
| (22) | $(S_1 \cap S_2)[a_1] \sqsubseteq S_3$ | **B2** | |
| (23) | $((S_1 \cap S_2) \cap S_3)[a_1] \sqsubseteq S_3$ | **PT** | (22),(4) |
| (24) | $((S_1 \cap S_2) \cap S_3) \sqsubseteq (S_1 \cap S_3)$ | **SI** | (5)(8) |
| (25) | $(S_1 \cap S_3)[a_2] \sqsubseteq S_3$ | **B2** | |
| (26) | $((S_1 \cap S_2) \cap S_3)[a_2] \sqsubseteq S_3$ | **PT** | (25),(24) |
| (27) | $((S_1 \cap S_2) \cap S_3)[(a_1 \cup a_2)] \sqsubseteq S_3$ | **AU** | (23),(26) |
| (28) | $((S_1 \cap S_2) \cap S_3)[A^{\Pi}] \sqsubseteq S_3$ | **AT** | (27),(1) |
| (29) | $((S_1 \cap S_2) \cap S_3)[A^{\Pi}] \sqsubseteq (S_1 \cap S_2)$ | **SI** | (14),(21) |
| (30) | $((S_1 \cap S_2) \cap S_3)[A^{\Pi}] \sqsubseteq ((S_1 \cap S_2) \cap S_3)$ | **SI** | (30),(28) |
| (31) | $((S_1 \cap S_2) \cap S_3) \sqsubseteq (((S_1 \cap S_2) \cap S_3) \cup \emptyset)$ | **UR** | |
| (32) | $((S_1 \cap S_2) \cap S_3)[A^{\Pi}] \sqsubseteq (((S_1 \cap S_2) \cap S_3) \cup \emptyset)$ | **ST** | (30),(31) |
| (33) | $(((S_1 \cap S_2) \cap S_3) \cap S_G^{\Pi}) \sqsubseteq ((S_1 \cap S_2) \cap S_3)$ | **IR** | |
| (34) | $(((S_1 \cap S_2) \cap S_3) \cap S_G^{\Pi}) \sqsubseteq (S_1 \cap S_2)$ | **ST** | (33),(4) |
| (35) | $(((S_1 \cap S_2) \cap S_3) \cap S_G^{\Pi}) \sqsubseteq S_G^{\Pi}$ | **IL** | |
| (36) | $(((S_1 \cap S_2) \cap S_3) \cap S_G^{\Pi}) \sqsubseteq ((S_1 \cap S_2) \cap S_G^{\Pi})$ | **SI** | (34),(35) |
| (37) | $((S_1 \cap S_2) \cap S_G^{\Pi}) \sqsubseteq \emptyset$ | **B1** | |
| (38) | $(((S_1 \cap S_2) \cap S_3) \cap S_G^{\Pi}) \sqsubseteq \emptyset$ | **ST** | (36),(37) |
| (39) | $(((S_1 \cap S_2) \cap S_3) \cap S_G^{\Pi})$ dead | **SD** | (2),(38) |

| # | judgment | rule | premises |
|---|---|---|---|
| (40) | $((S_1 \cap S_2) \cap S_3)$ dead | **PG** | (32),(2),(39) |
| (41) | $\{I^\Pi\} \sqsubseteq S_1$ | **B1** | |
| (42) | $\{I^\Pi\} \sqsubseteq S_2$ | **B1** | |
| (43) | $\{I^\Pi\} \sqsubseteq S_3$ | **B1** | |
| (44) | $\{I^\Pi\} \sqsubseteq (S_1 \cap S_2)$ | **SI** | (41),(42) |
| (45) | $\{I^\Pi\} \sqsubseteq ((S_1 \cap S_2) \cap S_3)$ | **SI** | (44),(43) |
| (46) | $\{I^\Pi\}$ dead | **SD** | (40),(45) |
| (47) | unsolvable | **CI** | (46) |

Table A.4: Translation of the 2-conjunctive certificate from Figure 4.5.

# B Detailed Coverage Results

This chapter contains detailed coverage results of variations of certifying Fast Downward. Tables B.1 and B.2 cover two variations of *certificate emitting* Fast Downward: $\mathrm{FD}^{\mathrm{C}'}$ avoids calling the cudd library by directly writing BDDs into a text file instead of storing them, while $\mathrm{FD}^{\mathrm{C}''}$ does not store a hint file. Table B.3 presents the results of $\mathrm{FD}^{\mathrm{P}'}$, a variation of *proof emitting* Fast Downward where set formulas are discarded as soon as they are not needed anymore in order to save memory usage.

| | $h^{\max}$ | | | $h^{\mathrm{M\&S}}$ | | |
|---|---|---|---|---|---|---|
| | FD | $\mathrm{FD}^{\mathrm{C}'}$ | $\mathrm{Ver}^{\mathrm{C}}$ | FD | $\mathrm{FD}^{\mathrm{C}'}$ | $\mathrm{Ver}^{\mathrm{C}}$ |
| 3unsat (30) | 15 | 10 | 10 | 15 | 11 | 10 |
| bag-barman (20) | 8 | 8 | 4 | 11 | 8 | 4 |
| bag-gripper (25) | 2 | 2 | 2 | 3 | 2 | 2 |
| bag-transport (29) | 7 | 6 | 4 | 7 | 6 | 4 |
| bottleneck (25) | 21 | 17 | 14 | 10 | 8 | 8 |
| cave-diving (25) | 7 | 7 | 6 | 7 | 7 | 6 |
| chessboard-pebbling (23) | 5 | 4 | 4 | 5 | 4 | 4 |
| diagnosis (13) | 5 | 5 | 5 | 4 | 4 | 4 |
| document-transfer (20) | 7 | 6 | 6 | 12 | 12 | 12 |
| mystery (9) | 2 | 1 | 1 | 2 | 2 | 1 |
| nomystery (150+24) | 59 | 37 | 25 | 60 | 47 | 38 |
| pegsol (24) | 24 | 24 | 24 | 24 | 24 | 24 |
| pegsol-row5 (15) | 5 | 5 | 4 | 5 | 5 | 4 |
| rovers (150+20) | 15 | 12 | 8 | 25 | 23 | 23 |
| sliding-tiles (20) | 10 | 10 | 10 | 10 | 10 | 10 |
| tetris (20) | 10 | 5 | 5 | 5 | 5 | 5 |
| tpp (25+30) | 23 | 15 | 12 | 37 | 36 | 30 |
| total (697) | 225 | 174 | 144 | 242 | 214 | 189 |

Table B.1: Completed tasks by domain for $\mathrm{FD}^{\mathrm{C}'}$.

| | $h^{\max}$ | | | $h^{\text{M\&S}}$ | | |
|---|---|---|---|---|---|---|
| | FD | $\text{FD}^{\text{C}''}$ | $\text{Ver}^{\text{C}}$ | FD | $\text{FD}^{\text{C}''}$ | $\text{Ver}^{\text{C}}$ |
| 3unsat (30) | 15 | 10 | 5 | 15 | 10 | 5 |
| bag-barman (20) | 8 | 8 | 0 | 11 | 8 | 0 |
| bag-gripper (25) | 2 | 2 | 2 | 3 | 2 | 2 |
| bag-transport (29) | 7 | 6 | 1 | 7 | 5 | 1 |
| bottleneck (25) | 21 | 17 | 12 | 10 | 8 | 4 |
| cave-diving (25) | 7 | 7 | 2 | 7 | 7 | 2 |
| chessboard-pebbling (23) | 5 | 4 | 2 | 5 | 4 | 2 |
| diagnosis (13) | 5 | 5 | 4 | 4 | 4 | 1 |
| document-transfer (20) | 7 | 6 | 5 | 12 | 12 | 12 |
| mystery (9) | 2 | 2 | 0 | 2 | 1 | 0 |
| nomystery (150+24) | 59 | 38 | 2 | 60 | 45 | 27 |
| pegsol (24) | 24 | 24 | 14 | 24 | 24 | 14 |
| pegsol-row5 (15) | 5 | 4 | 4 | 5 | 4 | 3 |
| rovers (150+20) | 15 | 12 | 1 | 25 | 24 | 18 |
| sliding-tiles (20) | 10 | 10 | 0 | 10 | 10 | 0 |
| tetris (20) | 10 | 5 | 5 | 5 | 5 | 5 |
| tpp (25+30) | 23 | 15 | 7 | 37 | 34 | 18 |
| total (697) | 225 | 175 | 66 | 242 | 207 | 114 |

Table B.2: Completed tasks by domain for $\text{FD}^{\text{C}''}$.

| | $h^{\max}$ | | | $h^{\text{M\&S}}$ | | |
|---|---|---|---|---|---|---|
| | FD | $\text{FD}^{\text{P}'}$ | $\text{Ver}^{\text{P}'}$ | FD | $\text{FD}^{\text{P}'}$ | $\text{Ver}^{\text{P}'}$ |
| 3unsat (30) | 15 | 10 | 10 | 15 | 10 | 10 |
| bag-barman (20) | 8 | 8 | 8 | 11 | 8 | 8 |
| bag-gripper (25) | 2 | 2 | 2 | 3 | 3 | 3 |
| bag-transport (29) | 7 | 6 | 6 | 7 | 6 | 6 |
| bottleneck (25) | 21 | 17 | 15 | 10 | 9 | 9 |
| cave-diving (25) | 7 | 7 | 7 | 7 | 7 | 7 |
| chessboard-pebbling (23) | 5 | 5 | 5 | 5 | 5 | 5 |
| diagnosis (13) | 5 | 5 | 5 | 4 | 4 | 4 |
| document-transfer (20) | 7 | 6 | 6 | 12 | 12 | 12 |
| mystery (9) | 2 | 1 | 1 | 2 | 2 | 1 |
| nomystery (150+24) | 59 | 31 | 30 | 60 | 45 | 37 |
| pegsol (24) | 24 | 24 | 24 | 24 | 24 | 24 |
| pegsol-row5 (15) | 5 | 5 | 5 | 5 | 5 | 5 |
| rovers (150+20) | 15 | 11 | 9 | 25 | 24 | 23 |
| sliding-tiles (20) | 10 | 10 | 10 | 10 | 10 | 10 |
| tetris (20) | 10 | 5 | 5 | 5 | 5 | 5 |
| tpp (25+30) | 23 | 15 | 14 | 37 | 33 | 29 |
| total (697) | 225 | 168 | 162 | 242 | 212 | 198 |

Table B.3: Completed tasks by domain for $\text{FD}^{\text{P}'}$.

# Bibliography

Abdulaziz, M., and Lammich, P. 2018. A formally verified validator for classical planning problems and solutions. In Alamaniotis, M., ed., *30th International Conference on Tools with Artificial Intelligence (ICTAI 2018)*, 474–479. IEEE.

Alcázar, V., and Torralba, Á. 2015. A reminder about the importance of computing and exploiting invariants in planning. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 2–6. AAAI Press.

Alkhazraji, Y.; Wehrle, M.; Mattmüller, R.; and Helmert, M. 2012. A stubborn set algorithm for optimal planning. In De Raedt, L.; Bessiere, C.; Dubois, D.; Doherty, P.; Frasconi, P.; Heintz, F.; and Lucas, P., eds., *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, 891–892. IOS Press.

Aspvall, B.; Plass, M. F.; and Tarjan, R. E. 1979. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters* 8(3):121–123.

Bäckström, C., and Nebel, B. 1995. Complexity results for $SAS^+$ planning. *Computational Intelligence* 11(4):625–655.

Bäckström, C.; Jonsson, P.; and Ståhlberg, S. 2013. Fast detection of unsolvable planning instances using local consistency. In Helmert, M., and Röger, G., eds., *Proceedings of the Sixth Annual Symposium on Combinatorial Search (SoCS 2013)*, 29–37. AAAI Press.

Bahar, R. I.; Frohm, E. A.; Gaona, C. M.; Hachtel, G. D.; Macii, E.; Pardo, A.; and Somenzi, F. 1993. Algebraic decision diagrams and their applications. In Lightner, M. R., and Jess, J. A. G., eds., *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 1993)*, 188–191.

Balyo, T., and Suda, M. 2016. Reachlunch entering the Unsolvability IPC 2016. In Muise, C., and Lipovetzky, N., eds., *Unsolvability International Planning Competition: planner abstracts*, 3–5.

Blanchette, J. C.; Fleury, M.; Lammich, P.; and Weidenbach, C. 2018. A verified SAT solver framework with learn, forget, restart, and incrementality. *Journal of Automated Reasoning* 61(1–4):333–365.

Bonet, B., and Castillo, J. 2011. A complete algorithm for generating landmarks. In Bacchus, F.; Domshlak, C.; Edelkamp, S.; and Helmert, M., eds., *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling (ICAPS 2011)*, 315–318. AAAI Press.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.

Bonet, B., and Helmert, M. 2010. Strengthening landmark heuristics via hitting sets. In Coelho, H.; Studer, R.; and Wooldridge, M., eds., *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, 329–334. IOS Press.

Brace, K. S.; Rudell, R. L.; and Bryant, R. E. 1990. Efficient implementation of a BDD package. In *Proceedings of the 27nd ACM/IEEE Conference on Design Automation (DAC 1990)*, 40–45.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 35(8):677–691.

Conchon, S.; Mebsout, A.; and Zaïdi, F. 2015. Certificates for parameterized model checking. In Bjørner, N., and de Boer, F., eds., *Proceedings of the 20th International Symposium on Formal Methods (FM 2015)*, 126–142. Springer, Cham.

Cook, S. A. 1971. The complexity of theorem-proving procedures. In Harrison, M. A.; Banerji, R. B.; and Ullman, J. D., eds., *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC 1971)*, 151–158. ACM New York.

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17:229–264.

Davis, M., and Putnam, H. 1960. A computing procedure for quantification theory. *Journal of the ACM* 7(3):201–215.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Communications of the ACM* 5(7):394–397.

Dechter, R., and Itai, A. 1992. Finding all solutions if you can find one. In *AAAI 1992 Workshop on Tractable Reasoning*, 35–39.

Edelkamp, S., and Helmert, M. 2001. The model checking integrated planning system (MIPS). *AI Magazine* 22(3):67–71.

Edelkamp, S., and Kissmann, P. 2009. Optimal symbolic planning with action costs and preferences. In Boutilier, C., ed., *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1690–1695. AAAI Press.

Edelkamp, S., and Kissmann, P. 2011. On the complexity of BDDs for state space search: A case study in Connect Four. In Burgard, W., and Roth, D., eds., *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*, 18–23. AAAI Press.

Edelkamp, S. 2001. Planning with pattern databases. In Cesta, A., and Borrajo, D., eds., *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, 84–90. AAAI Press.

Eriksson, S.; Röger, G.; and Helmert, M. 2017. Unsolvability certificates for classical planning. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 88–97. AAAI Press.

Eriksson, S.; Röger, G.; and Helmert, M. 2018a. Inductive certificates of unsolvability for domain-independent planning. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, 5244–5248. AAAI Press.

Eriksson, S.; Röger, G.; and Helmert, M. 2018b. A proof system for unsolvable planning tasks. In de Weerdt, M.; Koenig, S.; Röger, G.; and Spaan, M., eds., *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 65–73. AAAI Press.

Eriksson, S. 2019a. Code from the PhD thesis "certifying planning systems: Witnesses for unsolvability". https://doi.org/10.5281/zenodo.3355459.

Eriksson, S. 2019b. Experimental data from the PhD thesis "certifying planning systems: Witnesses for unsolvability". https://doi.org/10.5281/zenodo.3355471.

Eriksson, S. 2019c. Unsolvable PDDL benchmarks. https://doi.org/10.5281/zenodo.3355446.

Esparza, J.; Lammich, P.; Neumann, R.; Nipkow, T.; Schimpf, A.; and Smaus, J.-G. 2013. A fully verified executable LTL model checker. In Veith, N. S. H., ed., *Proceedings of the International Conference on Computer Aided Verification (CAV 2013)*, 463–478. Springer, Berlin, Heidelberg.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Gelder, A. V. 2002. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *Proceedings of the International Symposium on Artificial Intelligence and Mathematics (ISAIM 2002)*.

Gelder, A. V. 2008. Verifying RUP proofs of propositional unsatisfiability. In *Proceedings of the International Symposium on Artificial Intelligence and Mathematics (ISAIM 2008)*.

Gentzen, G. 1935. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift* 39(1):176–210.

Gnad, D.; Torralba, Á.; Hoffmann, J.; and Wehrle, M. 2016. Decoupled search for proving unsolvability. In Muise, C., and Lipovetzky, N., eds., *Unsolvability International Planning Competition: planner abstracts*, 16–18.

Goldberg, E., and Novikov, Y. 2003. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the conference on Design, Automation and Test in Europe – Volume 1 (DATE 03)*, 10886–10891. IEEE Computer Society Washington.

Gödel, K. 1929. *Über die Vollständigkeit des Logikkalküls*. Ph.D. Dissertation, University of Vienna.

Gödel, K. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik* 38(1):173–198.

Hammarberg, J., and Nadjm-Tehrani, S. 2005. Formal verification of fault tolerance in safety-critical reconfigurable modules. *International Journal on Software Tools for Technology Transfer* 7(3):268–279.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In Chien, S.; Kambhampati, S.; and Knoblock, C. A., eds., *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, 140–149. AAAI Press.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005)*, 1163–1168. AAAI Press.

Haslum, P. 2009. $h^m(P) = h^1(P^m)$: Alternative characterisations of the generalisation from $h^{\max}$ to $h^m$. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 354–357. AAAI Press.

Haslum, P. 2012. Incremental lower bounds for additive cost planning problems. In McCluskey, L.; Williams, B.; Silva, J. R.; and Bonet, B., eds., *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*, 74–82. AAAI Press.

Haslum, P. 2017. INVAL: the Independent PDDL plan Validator. https://github.com/patrikhaslum/INVAL. Accessed September 29, 2017.

Håstad, J. 1987. *Computational limitations of small depth circuits*. Ph.D. Dissertation, Massachusetts Institute of Technology.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM* 61(3):16:1–63.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 176–183. AAAI Press.

Helmert, M.; Röger, G.; and Sievers, S. 2015. On the expressive power of non-linear merge-and-shrink representations. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 106–114. AAAI Press.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hermann, M., and Pichler, R. 2010. Counting complexity of propositional abduction. *SIAM Journal on Computing* 76(7):634–649.

Heule, M.; Hunt, W. A.; and Wetzler, N. 2013a. Trimming while checking clausal proofs. In Jobstmann, B., and Ray, S., eds., *Proceedings of Formal Methods in Computer Aided Design (FMCAD 2013)*, 181–188. IEEE.

Heule, M. J. H.; Hunt, W. A.; and Wetzler, N. 2013b. Verifying refutations with extended resolution. In *Proceedings of the Twenty-Fourth International Conference on Automated Deduction (CADE 24)*, 345–359. Springer, Berlin, Heidelberg.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J.; Kissmann, P.; and Torralba, Á. 2014. "Distance"? Who cares? Tailoring merge-and-shrink heuristics to detect unsolvability. In Schaub, T.; Friedrich, G.; and O'Sullivan, B., eds., *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, 441–446. IOS Press.

Howey, R., and Long, D. 2003. VAL's progress: The automatic validation tool for PDDL2.1 used in the International Planning Competition. In Edelkamp, S., and Hoffmann, J., eds., *Proceedings of the ICAPS 2003 Workshop on the Competition: Impact, Organisation, Evaluation, Benchmarks*.

Järvisalo, M.; Heule, M. J.; and Biere, A. 2012. Inprocessing rules. In Gramlich, B.; Miller, D.; and Sattler, U., eds., *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2012)*, 355–370. Springer, Berlin, Heidelberg.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In Neumann, B., ed., *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 1992)*, 359–363. John Wiley and Sons.

Keyder, E.; Hoffmann, J.; and Haslum, P. 2014. Improving delete relaxation heuristics through explicitly represented conjunctions. *Journal of Artificial Intelligence Research* 50:487–533.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for and/or graphs. In Coelho, H.; Studer, R.; and Wooldridge, M., eds., *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, 335–340. IOS Press.

Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; and Winwood, S. 2009. seL4: Formal verification of an OS kernel. In Anderson, T., ed., *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 207–220. ACM New York, NY, USA.

Lammich, P. 2017. Efficient verified (UN)SAT certificate checking. In de Moura, L., ed., *Proceedings of the International Conference on Automated Deduction (CADE 2107)*, 237–254. Springer, Cham.

Lipovetzky, N.; Muise, C.; and Geffner, H. 2016. Traps, invariants, and dead-ends. In Coles, A.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2016)*, 211–215. AAAI Press.

Locher, R. 2019. Unsolvability proofs with non-linear merge-and-shrink heuristics. Master's thesis, University of Basel.

McConnell, R. M.; Mehlhorn, K.; Näher, S.; and Schweitzer, P. 2011. Certifying algorithms. *Computer Science Review* 5(2):119–162.

McCune, W. 1997. Solution of the Robbins problem. *Journal of Automated Reasoning* 19(3):263–276.

Miltersen, P. B.; Radhakrishnan, J.; and Wegener, I. 2005. On converting CNF to DNF. *Theoretical Computer Science* 347(1–2):325–335.

Muise, C. 2018. Characterizing and computing all delete-relaxed dead-ends. In *ICAPS 2018 Workshop on Constraint Satisfaction Techniques for Planning and Scheduling*, 29–34.

Pěnička, M. 2007. Formal approach to railway applications. In Jones, C. B.; Liu, Z.; and Woodcock, J. C. P., eds., *Formal Methods and Hybrid Real-Time Systems*. Berlin, Heidelberg: Springer. 504–520.

Rintanen, J. 2008. Regression for classical and nondeterministic planning. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, 568–572.

Robinson, J. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12(1):23–41.

Russinoff, D. M. 2000. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon™ processor. In Hunt, W. A., and Johnson, S. D., eds., *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, 22–55. Springer, Berlin, Heidelberg.

Schaefer, T. J. 1978. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (STOC '78)*, 216–226. New York: ACM Press.

Seipp, J.; Pommerening, F.; Sievers, S.; Wehrle, M.; Fawcett, C.; and Alkhazraji, Y. 2016. Fast Downward Aidos. In Muise, C., and Lipovetzky, N., eds., *Unsolvability International Planning Competition: planner abstracts*, 28–38.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. `https://doi.org/10.5281/zenodo.790461`.

Somenzi, F. 2015. CUDD 3.0.0. `https://github.com/ivmai/cudd`. Unofficial git mirror of `http://vlsi.colorado.edu/~fabio/`. Accessed: 10.02.2019.

Ståhlberg, S. 2017. *Methods for Detecting Unsolvable Planning Instances using Variable Projection*. Ph.D. Dissertation, Linköping University.

Steinmetz, M., and Hoffmann, J. 2017. State space search nogood learning: Online refinement of critical-path dead-end detectors in planning. *Artificial Intelligence* 245:1–37.

Suda, M. 2014. Property directed reachability for automated planning. *Journal of Artificial Intelligence Research* 50:265–319.

Torralba, Á.; Alcázar, V.; Borrajo, D.; Kissmann, P.; and Edelkamp, S. 2014. SymBA*: A symbolic bidirectional A* planner. In *Eighth International Planning Competition (IPC-8): planner abstracts*, 105–109.

Torralba, Á. 2015. *Symbolic Search and Abstraction Heuristics for Cost-Optimal Planning*. Ph.D. Dissertation, Universidad Carlos III de Madrid.

Valiant, L. G. 1979. The complexity of enumeration and reliability problems. *SIAM Journal on Computing* 8(3):410–421.

Valmari, A. 1989. Stubborn sets for reduced state space generation. In Rozenberg, G., ed., *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets (APN 1989)*, volume 483 of *Lecture Notes in Computer Science*, 491–515. Springer-Verlag.

Wehrle, M., and Helmert, M. 2012. About partial order reduction in planning and computer aided verification. In McCluskey, L.; Williams, B.; Silva, J. R.; and Bonet, B., eds., *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*, 297–305. AAAI Press.

Wehrle, M., and Helmert, M. 2014. Efficient stubborn sets: Generalized algorithms and selection strategies. In Chien, S.; Fern, A.; Ruml, W.; and Do, M., eds., *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 323–331. AAAI Press.