University of Basel

Certifying Unsolvability using CNF Formulas

Master thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
ai.dmi.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Salomé Eriksson

Fabian Kruse
fabian.kruse@unibas.ch
2018-057-356

February 07, 2023

Acknowledgments

First, I would like to express my gratitude to Prof. Dr. Malte Helmert for the opportunity to write this thesis in the Artificial Intelligence research group.
Secondly, I want to thank Dr. Salomé Eriksson for her help and supervision. Her guidance and advice carried me through all stages of writing my thesis.
And finally, I would like to thank my family and girlfriend for their unconditional and continuous support.

Abstract

Certifying algorithms is a concept developed to increase trust by demanding affirmation of the computed result in form of a certificate. By inspecting the certificate, it is possible to determine correctness of the produced output. Modern planning systems have been certifying for long time in the case of solvable instances, where a generated plan acts as a certificate.

Only recently there have been the first steps towards certifying unsolvability judgments in the form of *inductive certificates* which represent certain sets of states. Inductive certificates are expressed with the help of propositional formulas in a specific formalism.

In this thesis, we investigate the use of propositional formulas in *conjunctive normal form* (CNF) as a formalism for inductive certificates. At first, we look into an approach that allows us to construct formulas representing inductive certificates in CNF. To show general applicability of this approach, we extend this to the family of delete relaxation heuristics. Furthermore, we present how a planning system is able to generate an *inductive validation formula*, a single formula that can be used to validate if the set found by the planner is indeed an inductive certificate. At last, we show with an experimental evaluation that the CNF formalism can be feasible in practice for the generation and validation of inductive validation formulas.

Table of Contents

Acknowledgments **ii**

Abstract **iii**

1 Introduction **1**

2 Related Work **3**

3 Background **6**

3.1 Propositional Logic 6

3.2 Planning Tasks 7

3.3 Inductive Certificate 10

4 Methods **12**

5 Theory **15**

5.1 Queries 15

5.2 Efficiency despite SAT 17

6 Application to search algorithms **18**

6.1 Blind Search 19

6.2 Heuristic Search 20

6.2.1 Delete Relaxation 21

7 Validation Formulas **24**

7.1 Generating Validation Formulas 24

7.2 Verifying Validation Formulas 26

7.2.1 Transforming the formula 28

7.2.2 Tseitin Transformation 29

8 Experimental Setup **32**

8.1 SAT-Solver 33

9 Experimental Results **34**

9.1 Implementation 34

9.2 Generation 36

9.3 Verification 40

9.4 Summary 40

10 Comparison **42**
10.1 Generation . . . 42
10.2 Verification . . . 44
10.3 Summary . . . 45

11 Conclusion **47**
11.1 Future Work . . . 48

Bibliography **49**

1 Introduction

Planning is a research field in the area of artificial intelligence that investigates different ways to solve a very natural problem: assume that we are given a starting point called the *initial state*, a set of actions that enables us to transition between different states, and a goal description. Given that problem setting, planning tries to answer the question which sequence of actions allows to us to get to the goal from the initial state. There are many variations of that question where we take other factors into consideration: do actions potentially fail? What can we do if not everything is observable?

For this work, we will focus on *classical planning*, where we assume everything to be *deterministic* and *fully observable*, so if we apply an action, we know how the state will change.

Planning systems are highly developed algorithms that try to generate a sequence of actions called *plan*, a solution for a given problem description. Generally, planning systems can give one of two answers when presented a problem: either, a solution to the problem in the form of a plan, or a statement declaring the task to be unsolvable. If the planning system finds a solution to a problem, it is rather easy to check if the system did make any mistakes: since the solution consists of a sequence of actions, we can simply test the correctness of a plan by applying it to the initial state. If we would end up in a state that fits the goal description, the planning system produced a valid solution to the problem. However, it is much more difficult to validate a statement of the form "the task is impossible to solve".

For that reason, Eriksson, Röger, and Helmert (2017) developed an approached to augment planning systems to be capable of producing a reason for its unsolvability judgement. These reasons are called *inductive certificates* and are produced in a way such that they can be validated independently. In that sense, inductive certificates act as a counterpart to plans in the case of unsolvable tasks.

Further, Eriksson, Röger, and Helmert (2017) proposed a list of properties that help to analyze the suitability of certificates: there should exist a certificate if and only if the plan is unsolvable, certificates should be efficient to generate and validate, and we would like to extend many planning systems to produce certificates. As inductive certificates reason about sets of states, a central problem is the representation of these sets.

In this work, we will develop a procedure to generate certificates represented as CNF formulas, where we will try to take these properties into consideration. The choice of using CNF formulas is on one hand very natural, as problems in itself are often described with propositional logic, and on the other hand, we already have a wide range of verification tools available: we will formulate certificates in a way such that they can be verified by arbitrary SAT-solvers.

However, CNF-formulas have a well-known drawback: we don't have a theoretical guarantee for their efficiency regarding satisfiability checks, which are essential for the verification of the certificates. Therefore, we will investigate how well CNF formulas are suited for certificate representation in practice, under the condition of allowing this inefficiency.

2 Related Work

One of the main goals of computer science is to develop algorithms that produce a correct answer for a given problem.Whereas the theoretical side of computer science usually focuses on formally proving algorithms to work correctly, practical implementations can often only be evaluated partially and empirically. A very natural idea to enhance trust in an algorithm was presented by McConnell et al. (2011): they proposed the concept of *certifying algorithms.* These are algorithms that produce "with each output a certificate or witness that the particular output has not been compromised by a bug." (McConnell et al., 2011, p.1). The presented output together with the certificate can then be inspected to determine whether the algorithm worked correctly or not. In that sense, it is possible to validate the correctness of the presented answer independently.

However, certifying the output of an algorithm is not a modern concept: the famous *Euclidean algorithm* can be used to determine the greatest common divisor g of two inputs x and y. The algorithm can easily be modified to not only calculate the number g, but also to generate two additional numbers a and b with the property $g = xa + yb$. This version of the algorithm produces a certificate in the form of a and b for the generated output value g: the output g can be validated easily by testing if $g = xa + yb$ and g divides both a and b. Although not completely trivial, this guarantees that g is not just any, but indeed the greatest common divisor of x and y.

Classical planning intends to find a plan for a given planning task or to prove that the task is unsolvable. The planning community mostly focused on algorithms that find and generate plans if existent and therefore the problem of proving a task unsolvable stayed, for the most part, relatively unattended. The *International Planning Competition* (IPC) is part of the *International Conference on Automated Planning and Scheduling* (ICAPS), one of the largest conferences on planning. Different planning systems can compete against each other in the competition by evaluating a large number of planning problems. The IPC has several tracks, each with different focus. The negligence of the problem of proving tasks unsolvable is reflected in the fact that until 2014 almost all problems in the IPC are solvable. Only first efforts that target unsolvable problems, such as an algorithm to detect unsolvable planning tasks as fast as possible (Bäckström, Jonsson, and Ståhlberg, 2013), brought a change to this. With an upcoming interest of the community in unsolvable planning tasks, the IPC instantiated a separate competition in 2016 that focuses purely on unsolvable tasks. Although the problem of detecting unsolvability has made significant advances in recent years, there has been little investigation into the problem of verifying unsolvability claims.

Eriksson, Röger, and Helmert, 2017 made a first step in the development of fully certifying planning systems. They initiated the idea of generating certificates using *inductive sets.*

Informally, these are sets of states that can never be left by any action application. This means that once a state in an inductive set is reached, every action leads to a state in the same set. With the help of these sets, we can formulate unsolvability of a planning task as follows: if an inductive set of states contains the initial state and no goal state, then by inductivity of the set, there cannot exist a sequence of actions that lead to a goal state and hence the task must be unsolvable. Further, the inductive set acts as a certificate itself and is hence fittingly called *inductive certificate*. Additionally, Eriksson, Röger, and Helmert (2017) showed that the existence of such a specific inductive set is equivalent to the task being unsolvable. This means, that once we found such a set of states, the task must be unsolvable. The inductive set often consists out of a huge number of states, and hence the performance of the approach heavily relies on how to represent the inductive set. The authors discussed three formalism, namely 2-CNF, Horn-formulas and BDD's, and presented a practical evaluation of the developed methods. These three formalisms were chosen, because they efficiently perform operations that are required to verify that the described set indeed has the property of an inductive certificate.

Another formalism that is widely studied is given by propositional formulas in conjunctive normal form (CNF). Although CNF formulas are commonly used in many areas of computer science, they have a drawback: checking CNF formulas for satisfiability is NP-complete and even described as "a problem of central importance in computer science"(Vardi, 2014, p.1). Thus, it does not seem to be a good formalism for inductive certificates at first glance.

Despite this, there is evidence that in practice satisfiability checking is often feasible: "[...]have designed and implemented highly scalable CDCL SAT solving algorithms (or simply, SAT-solvers) that are able to efficiently solve multi-million variable instances obtained from real-world applications" (Ganesh and Vardi, 2020, p.2). There has been "remarkable success" (Biere, Heule, and Maaren, 2009, p.146) in the use of SAT-solvers for CNF formulas generated from practical applications. In that sense, "solvers are efficient for many classes of large real-world instances" (Ganesh and Vardi, 2020, p.1) and the current belief hints towards a beneficial structure of CNF formulas that are based on practical examples in contrast to randomly generated formulas.

Hence, it might be interesting to investigate whether validating inductive certificates based on CNF formulas can be feasible, since planning tasks are naturally structured by design of the problem description.

Certifying algorithms naturally spark a spiral of certificates: a first algorithm emits a certificate, which needs to be validated in a validation tool. However, this simply shifts the problem to the validator, as we are left again to simply trust it or require it to be certifying on its own. This creates a chain of certifying validation tools until at one point we decide to trust the validation algorithm. This observation provides us with another argument for the use of CNF formulas to certify unsolvable planning systems.

The SAT-community has a long-lasting history with certifying algorithms initiated by Goldberg and Novikov (2003), in which the authors describe an efficient procedure to generate unsatisfiability proofs and a corresponding validation technique. The authors proposed a method in which the SAT-solver emits an unsolvability certificate as a series of unit propagation that are part of the search process already and allow validating the solver's work. Although this first version of certifying SAT-solvers was not performant enough to be used in practice, it instituted the idea of certifying unsatisfying outputs in the SAT community. In fact, providing unsolvability certificates has become the standard for SAT-solvers: the SAT competition[1], a competitive event for the Boolean satisfiability problem,

[1] http://www.satcompetition.org/

requires from 2013 onwards all algorithms that classify a formula as unsatisfiable to also provide a certificate.

The progress in the SAT community regarding certifying algorithms provide an additional reason to use CNF formulas to certify planning systems: instead of developing a certifying verifier on our own, we only have to translate the unsolvability argument into SAT and have a range of certifying verifiers available.

3 Background

In this chapter, we will introduce important terminology used in this work. At first, we introduce propositional logic, which allows us to define core concepts in the field of classical planning. Finally, we look into inductive certificates.

3.1 Propositional Logic

Propositional logic is a concept that describes how we can express logical conditions with formulas. Its most basic element is a simple propositional variable v. This variable can only have one of two values: *true* or *false*. Further, we can connect proportional formulas with the logical "and" $\land$ and the logical "or" $\lor$ called conjunction and disjunction respectively. This allows us to create more complex formulas.
The following will describe how we can use propositional formulas to express logical relations. We make use of them extensively to represents individual and sets of states. Propositional formulas can be constructed as follows:

Definition 1 (propositional formula)**.** Propositional formulas *are defined over a set of propositional variables V according to the following rules:*

- *For all $v \in V$, v is a propositional formula*
- *If φ is a propositional formula, then $\neg\varphi$ is a propositional formula*
- *If φ and ψ are propositional formulas, then $(\varphi \lor \psi)$ and $(\varphi \land \psi)$ are propositional formulas*

Further, we use abbreviation $(\varphi \rightarrow \psi)$ called implication *for $(\neg\varphi \lor \psi)$ and $(\varphi \leftrightarrow \psi)$ called* equijunction *for $((\varphi \rightarrow \psi) \land (\psi \rightarrow \varphi))$.*

Although, propositional formulas can be used to express a certain condition, they only have an actual meaning if there are interpreted. Propositional formulas are interpreted as follows:

Definition 2 (interpretation, model)**.** *Let V be a set of propositional variables.*
An interpretation *of V is a function $I : V \to \{\top, \bot\}$.*
We say that I satisfies φ, or is a model of φ, written as $I \models \varphi$ if:

- *$\varphi = v$ and $I(v) = \top$ for all $v \in V$*
- *$\varphi = \neg\psi$ and $I \not\models \psi$*
- *$\varphi = \psi \vee \psi'$ and $I \models \psi$ or $I \models \psi'$*
- *$\varphi = \psi \wedge \psi'$ and $I \models \psi$ and $I \models \psi'$*

When dealing with propositional formulas, we are often interested in how they relate to each other. A very important property describes when two propositional formulas are *equivalent*.

Definition 3 (equivalent)**.** *Let φ and ψ be propositional formulas over propositional variables in V.*
The formulas φ and ψ are equivalent *iff for every Interpretation I we have $I \models \varphi$ iff $I \models \psi$.*
In particular, the formulas φ and ψ have the same set of models.

Often it is useful to consider formulas with a certain structure, so-called *normal forms*. To this end, we introduce some additional terminology commonly used: the most basic propositional formulas are called *literals* and consist of only a single, possibly negated, variable. Connecting a finite amount of literals with conjunctions and disjunctions yields a *clause* or *cube*.

Definition 4 (literal, clause, cube)**.** *Let p be a propositional variable. The variable p and its negation $\neg p$ are called* literals*. Any finite collection over literals that is connected over a disjunction of variables in V is called a* clause*.*
Contrary, a finite collection of literals over a conjunction is called a cube*.*

The main normal form considered in this work is the *conjunctive normal form* defined as follows:

Definition 5 (conjunctive normal form)**.** *A formula φ is in* conjunctive normal form (CNF) *iff it is a conjunction of a finite number of clauses.*

3.2 Planning Tasks

A classical planning task can be understood as a description of a problem consisting of four components: a set of state variables which implicitly define the set of all states, an initial state of the problem setting, a set of actions that can be used to alter between states and a goal description we want to satisfy.
Before we begin to introduce the different concepts used in this work, we will have a look at our running example. For this, we will consider an instance of a Sokoban puzzle presented in Figure 3.1. The Sokoban puzzle describes a problem in which a man needs to move a box from `B3` to `B5`. However, the man can only push the box and is unable to pull it.
For this thesis, we will focus on planning tasks formulated in propositional STRIPS representation Fikes and Nilsson, 1971, where we represent a *state* by the set of true variables. This is particularly useful, because we can easily generate propositional formulas that represent sets of states. Expressing states with propositional formula enables us to not only represent states compactly, but also to make use of powerful tools such as transformation algorithms and SAT-solvers.
The Sokoban puzzle, for example, could have a variable `man-at-B2` expressing if the man is currently at position `B2`.

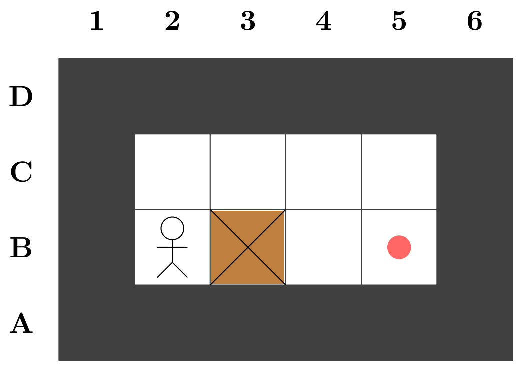


Figure 3.1: An instance of the Sokoban puzzle. Dark cells are walls, the brown cell is the position of the box, and the red dot marks the goal position

A *state* is described by a specific configuration of propositional variables, in our case denoted by the set of true variables:

Definition 6 (state)**.** *Let V be a set of propositional variables. A subset $s \subseteq V$ is called a* state *(over V).*

We could represent the Sokoban puzzle with the variables that describe the position of the box `box-at-*` and the position of the man `man-at-*` for all grid positions $\star$. In that case, the set V consists out of all these variables. The state presented in Figure 3.1 can then be expressed with the set `{man-at-B2, box-at-B3}`.
The description of a goal is given by a set of variables. A state is a goal state, if the goal description is true in that state. In particular, this allows for multiple states to be a goal state. We could, for instance, express the goal of the Sokoban task with the set `{box-at-B5}` and hence for example both `{man-at-B3,box-at-B5}` and `{man-at-C4, box-at-B5}` are goal states.

Definition 7 (goal state)**.** *Let V be a set of propositional variables and $G \subseteq V$ the goal. A state $s \subseteq V$ is called* goal state *if $G \subseteq s$.*

The STRIPS formalism allows actions to change values of certain variables. Therefore, actions can be used to switch between different states.
An action is a tuple of three different sets of variables, each with a specific role. The set of preconditions contains variables that need to be fulfilled by a state for the action to be applicable in that state. Further, an action contains a set of add- and delete-effects describing which variables are added and removed by applying that action in a state.

Definition 8 (action, action applicability)**.** *Let V be a set of propositional variables, s be a state. We call the tuple $a = \langle pre(a), add(a), del(a)\rangle$ an* action*, where $pre(a) \subseteq V$ is the* precondition*, $add(a) \subseteq V$ the* add-effects *and $del(a) \subseteq V$ the* delete-effects *of a.*
Further, we say that a is applicable *in s iff $pre(a) \subseteq s$. Applying a to s results in the successor state $s[a]$ defined as:*

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

A sequence of actions $\pi = \langle a_1, \dots a_n\rangle$ is applicable in s iff $s[a_1]\dots[a_n]$ is well-defined for all $1 \leq i \leq n$. The resulting state after applying the action sequence π to state s is denoted by $s[\pi]$.

We could for example define the following action `push-B3-B4` that encodes the act of pushing the box from `B3` to `B4`:

$$\texttt{push-B3-B4} := \langle \{\texttt{box-at-B3}, \texttt{man-at-B2}\}, \{\texttt{box-at-B4}, \texttt{man-at-B3}\},$$
$$\{\texttt{box-at-B3}, \texttt{man-at-B2}\} \rangle$$

Naturally, this is only possible if the box is actually at position `B3` and the man is in the position behind the box, otherwise, he would not be able to push the box. This is expressed in the precondition-set. The effect-sets of the action come naturally, as pushing the box from `B3` to `B4` changes the location of the box and the man.
Now we have all tools available to define the centerpiece of classical planning: a planning task.

Definition 9 (STRIPS planning task)**.** *A* STRIPS planning task Π *is defined as* $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$ *where*

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

Further, S^Π *denotes the set of all states of* Π *and* S^Π_G *the set of all goal states.*

Typically, actions have an associated cost specified by a cost function, representing the cost of applying that action. Since action costs have no influence on the unsolvability of planning tasks, the action cost function is negligible for our purposes.
Classical planning aims to find an action sequence that leads from the initial state to a goal state. This action sequence is called a *plan* for planning task Π and is one solution of possibly many for a planning task.
To illustrate, we could define the action sequence $\pi = \langle$ `push-B3-B4`, `push-B4-B5` $\rangle$ for our Sokoban puzzle. After applying the actions contained in π to the initial state depicted in Figure 3.1, we would have reached a goal state as the box would be at position `B5`.

Definition 10 (plan)**.** *Let* Π *be a planning task. An action sequence* $\pi = \langle a_1, \ldots, a_n \rangle$ *is called a* plan *if* π *is applicable in* I^Π *and* $G \subseteq I^\Pi[\pi]$*, in other words:* π *is called a plan if it defines a sequence of actions that lead from the initial state* I^Π *to a goal state.*

It is often possible to find a plan for a given planning task. In that case, the plan is considered to be a solution to the planning task and hence the planning task is *solvable*. However, this is not always the case, and naturally the planning task is then called *unsolvable*.
In this work, we will often argue about sets of states, and therefore we generalize our notion of action applicability.

Definition 11 (progression)**.** *Let* Π *be a planning task,* $S \subseteq S^\Pi$ *a set of states and* $a \in A^\Pi$ *an action. The set* $S[a] = \{s[a] \mid s \in S, a \text{ applicable in } s\}$ *is called the* progression *of* S *with* a*.*
Similarly, let $A \subseteq A^\Pi$ *be a set of actions. The progression of* S *with* A *is given by* $S[A] = \bigcup_{a \in A} S[a]$*.*

Furthermore, we sometimes like to traverse the search space in opposite direction. This process is called regression, which makes use of backwards-applicable actions.

Definition 12. *Let s be a state and a an action.*
We say that a is backwards-applicable *in s iff there exists a state s' with $s'[a] = s$.*
Applying an action in different states might lead to the same successor state.
Therefore, the predecessor of a state s via an action a is described by a set of states.
The set of predecessors of state s via action a is defined as:

$$[a]s = \{s' \mid a \textit{ is applicable in } s', s'[a] = s\}$$

Definition 12 allows us to define the regression of a set of states via actions.

Definition 13 (regression)**.** *Let Π be a planning task, $S \subseteq S^{\Pi}$ a set of states and $a \in A^{\Pi}$ an action. The set $[a]S = \{s' \mid a \textit{ applicable in } s', s'[a] \in S\}$ is called the* regression *of S with a.*
Similarly, let $A \subseteq A^{\Pi}$ be a set of actions. The regression of S with A is given by $[A]S = \bigcup_{a \in A}[a]S$.

3.3 Inductive Certificate

The basis for unsolvability certificates presented in this work is the concept of *inductive sets*, which are state sets closed under action application.

Definition 14 (inductive set)**.** *Let Π be a planning task and $S \subseteq S^{\Pi}$ a set of states. The set S is called* inductive *in Π if $S[A^{\Pi}] \subseteq S$, that is, all action applications in a state in S lead to a state in S.*

Inductive sets are impossible to leave, since if a state in S is reached, all further action applications still stay in S. This property makes them useful for unsolvability certificates, because if an inductive set contains no goal state, it would be impossible to reach a goal state from any state in the set. In particular, if the inductive set contains the initial state and no goal state, there can not exist a plan for the planning task. This is summarized in the definition of an *inductive certificate.*

Definition 15 (inductive certificate)**.** *Let Π be a planning task. An* inductive certificate *for Π is given by a set $S \subseteq S^{\Pi}$ of states, such that*

1. *$I^{\Pi} \in S$*
2. *$S \cap S_{G^{\Pi}} = \emptyset$*
3. *S is inductive in Π*

To highlight the difference between inductive certificates and the upcoming backwards inductive certificates, inductive certificates will sometimes be called *forward inductive certificates.*
Further, the notion of inductive sets can be extended to a regression perspective. Here, inductive sets describe sets that cannot be entered from the outside.

Definition 16 (backwards inductive set)**.** *Let Π be a planning task and $S \subseteq S^{\Pi}$ a set of states. The set S is called* backwards inductive *iff for all states $s \in S$ and backwards-applicable actions a we have $[a]s \subseteq S$*

A crucial topic for inductive certificates is their representation. Because the set of states required for the certificates tend to be very large, we would like to represent them compactly and efficiently. For this thesis, we will use propositional *CNF formulas* to represent an inductive certificate.
Since inductive certificates are defined as a set of states, we first need to observe how to relate propositional formulas and state sets. Propositional formulas describe a logical condition. If a state satisfies this condition, then they are a member of the set described by the formula.

Definition 17 (states of φ)**.** *Let φ be a propositional formula over the variables V^{Π} of a STRIPS planning task Π. The formula φ represents the set $states(\varphi)$ of states of Π as:*

$$states(\varphi) = \{s_{\mathcal{I}} \mid \mathcal{I} : V^{\Pi} \rightarrow \{\top, \bot\}, \mathcal{I} \models \varphi\}, \text{ where } s_{\mathcal{I}} = \{v \mid \mathcal{I}(v) = \top\} \tag{3.1}$$

For instance, consider the formula $\varphi = ((\neg v \wedge w) \vee q)$ over the set of variables $V = \{v, w, q\}$. By examining the models of formula φ, we see that for example the states $\{w\}, \{q\}$ and $\{w, q\}$ are states represented by φ, whereas for example the state $\{v, w\}$ is not. Notice that any variable that is not occurring in a formula can be assigned either $\top$ or $\bot$. If we consider the formula $\varphi = v$ over $V = \{v, w\}$, then $states(\varphi) = \{\{v\}, \{v, w\}\}$.
With the help of Definition 17, we can relate state sets S and S' represented by formulas φ and ψ as follows: $S \subseteq S'$ iff $\varphi \models \psi$.
Further, if a formula φ describes a set of states that is an inductive certificate, we call it an *inductive certificate formula.*

Definition 18 (inductive certificate formula)**.** *Let Π be a STRIPS planning task. An* inductive certificate formula *for Π is a propositional formula φ with $vars(\varphi) \subseteq V^{\Pi}$ such that $states(\varphi)$ is an inductive certificate for Π.*

To validate the generated certificate, we will need to formulate the progression or regression of a set of states. Symbolic search and SAT-planning introduce additional variables $V^{\Pi'} = \{v' \mid v \in V^{\Pi}\}$ to define a transition relation τ_a over the variables in $V^{\Pi} \cup V^{\Pi'}$ for each action $a \in A^{\Pi}$:

$$\tau_a = \bigwedge_{v_p \in pre(a)} v_p \wedge \bigwedge_{v_a \in add(a)} v'_a \wedge \bigwedge_{v_d \in (del(a) \setminus add(a))} \neg v'_d \wedge \bigwedge_{v \in V^{\Pi} \setminus (add(a) \cup del(a))} (v \leftrightarrow v') \tag{3.2}$$

4 Methods

For the following, we consider a planning task $\Pi = \langle V^\Pi, I^\Pi, A^\Pi, G^\Pi \rangle$, where we assume that Π is unsolvable. Further, let S be an inductive certificate for Π, so S is an inductive set of states that contains the initial state and no goal state. Inductivity of S implicitly partitions the search space in two areas: a first area that contains all states in S consisting of all reachable states and the complement of S, denoted by $\overline{S}$, describing a set of states that are unreachable. Similar to describing S as a set of states that can not be left, we can describe $\overline{S}$ informally as a set of states that can not be entered, i.e. there exists no path from the initial state I^Π to a state $\overline{s} \in \overline{S}$. Similar arguments as before give rise to a second class of inductive certificates called *backwards inductive certificates.*

Definition 19 (backwards inductive certificates)**.** *Let* Π *be a planning task. A* backwards inductive certificate *for* Π *is given by a set* $S_{back} \subseteq S^\Pi$ *of states such that*

1. $I^\Pi \notin S_{back}$
2. $S_{G^\Pi} \subseteq S_{back}$
3. S_{back} *is backwards inductive in* Π

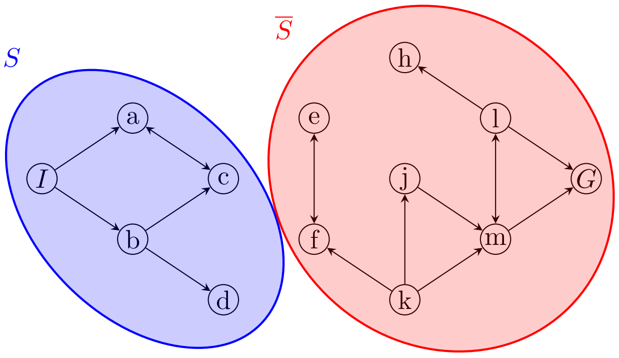


Figure 4.1: Forward and backwards inductive certificate for an unsolvable task

Figure 4.1 shows the graph of search space induced by an unsolvable planning task. Notice that S is an inductive certificate since it is inductive, contains the initial state I and no goal state (in this case G). The existence of S directly implies unsolvability of the underlying planning task, as shown in Proposition 1 in Eriksson, Röger, and Helmert (2017).
Similar, we can see that $\overline{S}$ is a backwards inductive certificate according to Definition 19. In the following, we will establish that the existence of $\overline{S}$, is equivalent to unsolvability of the task as well. Additionally, we will show that it is always possible to obtain a backwards inductive certificate from a forward inductive certificate. As demonstrated by Eriksson (2019a), inductive sets and backwards inductive sets are closely related.

Theorem 1 (Eriksson, 2019a, Theorem 4.2)**.** *Let S be a set of states. S is inductive iff $\overline{S}$ is backwards inductive.*

The following theorem shows that backwards inductive certificates can be used to certify unsolvability of a planning task.

Theorem 2 (soundness and completeness of backwards inductive certificates)**.** *Given a STRIPS planning task $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$. There exists a backwards inductive certificate for Π iff Π is unsolvable*

Proof. We need to show soundness and completeness of backwards inductive certificates:
Soundness: Assume that there exists a backward inductive certificate S_{back} for planning task Π. By Theorem 1, we have that $S_{ind} := \overline{S_{back}}$ is forward inductive. Further, from $G^\Pi \subseteq S_{back}$ it follows that $S_{ind} \cap G^\Pi = \emptyset$ and because of $I^\Pi \notin S_{back}$ we have $I^\Pi \in S_{ind}$. Therefore, S_{ind} is a forward inductive certificate. Notice that Eriksson, Röger, and Helmert (2017) showed that there exists a forward inductive certificate iff a planning task is unsolvable, which directly implies unsolvability of Π.
Completeness: Assume that planning task Π is unsolvable. To prove completeness, we have to show the existence of a backward inductive certificate according to Definition 19. As before, we use the result of Eriksson, Röger, and Helmert (2017) and deduce the existence of a forward inductive certificate S_{ind}. In the following, we will show that $\overline{S_{ind}}$ is a backwards inductive certificate.

1. Because S_{ind} is an inductive certificate, we have that $I^\Pi \in S_{ind}$. In particular, we have that $I^\Pi \notin \overline{S_{ind}}$
2. By definition of a planning task we know that S^{G^Π} is always non-empty and since S_{ind} is inductive, we have that $S_{ind} \cap S^{G^\Pi} = \emptyset$. Therefore, the set S^{G^Π} must be fully contained in $\overline{S_{ind}}$.
3. Since S_{ind} is inductive, it follows with from Theorem 1 that $\overline{S_{ind}}$ is backwards inductive in Π.

Therefore, $\overline{S_{ind}}$ is a backwards inductive certificate for Π. □

Theorem 2 proofs that backwards inductive certificates are sound and complete by themselves. However, this is not a surprise: the definition of a set complement gives us that all states which are not a set are in its complement and vice versa. Concluding with the results from Theorem 1, we can establish that a set is an inductive certificate iff its complement is a backwards inductive certificate.

Theorem 3 (complement of certificate is certificate)**.** *Given a STRIPS planning task* $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$ *and* S *a set of states.* S *is an inductive certificate iff* $\overline{S}$ *is a backwards inductive certificate.*

Proof. Let S be an inductive certificate for Π. By Definition 15 the set S has the following properties.

1. $I^\Pi \in S$
   By basic set operations, this holds iff $I^\Pi \notin \overline{S}$.

2. $S \cap G^\Pi \subseteq \emptyset$
   Again, by basic set operations, this holds iff $G^\Pi \subseteq \overline{S}$.

3. S is an inductive set
   By Theorem 1, we have that S is inductive iff $\overline{S}$ is backwards inductive.

In particular, $\overline{S}$ is a backwards inductive certificate according to Definition 19 iff S is an inductive certificate. □

5 Theory

In this chapter, we will investigate the use of CNF formulas to formalize inductive certificates. At first, we will look into different queries that we require to be supported efficiently by CNF formulas. This is particularly important from a theoretical standpoint, as it will form the base argument for a practical use of the formalism. In this part, we will be faced with the fact that satisfiability checks for CNF formulas are **NP**-complete and hence we have to adjust our definition of efficiency. In a second step, we will look into how uninformed search algorithms such as *blind search* can generate inductive certificates. Since we generally can not directly obtain a formula in CNF, we will investigate different approaches that allow us to derive a final CNF formula.

5.1 Queries

One of the core aspects of inductive certificates is the way of choosing a formalism that is able to represent huge sets of states compactly. There is a wide range of different formalisms available, such as BDD's and Horn formulas, all with their respective advantages and disadvantages. BDD's for example allow a compact and canonical representation that supports bounded disjunctions and conjunctions, but not arbitrary conjunctions. The difficulty arises from choosing a suitable representation that allows reasonably fast operations.

To make use of a formalism, it should be possible to support a range of different query operations needed for construction and verification of the certificates. For this, we consider a formalism to be suitable for certificate representation if it supports certain operations efficiently. We will evaluate efficiency under the following notion: a formalism supports an operation *efficiently* if it can perform the operation in time polynomial in the size of the formula.

This thesis will focus on formulating certificates using CNF formulas. Further, we consider the STRIPS formalism for formulating a planning task, which also makes use of proportional variables taking on only the values *true* or *false*. As we have seen in Definition 17, we can easily obtain a set of states defined by a specific propositional formula φ by investigating the models of φ. CNF are a particular instance of propositional formulas, and hence we can also represent state sets in CNF. Although CNF formulas generally do not allow efficient testing whether a given assignment is satisfying, they support useful operations such as model testing, renaming and conjunction of literals efficiently. Furthermore, because of the popularity of SAT-competitions, there is a wide range of SAT-solvers available for verification.

In the following, we will analyze different queries and whether CNF formulas can efficiently support it. The list of queries is based on Theorem 4.4 in Eriksson (2019a), which establishes that inductive certificates can be verified efficiently for any formalism, if the formalism efficiently supports **MO**, **CE**, **SE**, $\wedge$**BC**, **CL** and $\mathbf{RN}_{\prec}$, which are defined and explained below.
Formula φ is considered to be an instance of a propositional formula over a variable set V, and the corresponding set $vars(\varphi)$ describes the variables occurring in formula φ. The size of the representation is denoted by $|| \varphi ||$.
The results are taken from Darwiche and Marquis (2002). Additionally, we provide a brief reasoning for each operation.

- Model Testing (**MO**)
  Given formula φ and truth assignment $\mathcal{I}$, test if $\mathcal{I} \models \varphi$

  We can test whether $\mathcal{I} \models \varphi$ by simply evaluating the interpretation $\mathcal{I}$. As with any propositional formula, evaluating an assignment can be done in linear time.
  Therefore, **MO** is supported efficiently.

- Consistency (**CO**)
  Given formula φ, test whether φ is satisfiable

  As mentioned earlier, **CO** describes the famous SAT problem. We will discuss what this means for our efficiency analysis in Chapter 5.2.
  CO is not supported efficiently.

- Clausal Entailment (**CE**)
  Given formula φ and clause ϕ, test whether $\varphi \models \phi$

  Notice, that $\varphi \models \phi$ is equivalent to testing whether the CNF formula $\varphi \wedge \neg\phi$ is inconsistent.
  Since **CO** is not supported, **CE** is not efficiently supported either.

- Sentential Entailment (**SE**)
  Given formulas φ and ψ, test whether $\varphi \models \psi$

  Since ψ is a CNF formula, $\psi = \bigwedge_i c_i$, where c_i are the clauses of ψ. Then $(\varphi \models \psi)$ is equivalent to $(\varphi \wedge \neg\psi)$ being inconsistent. Therefore, $(\varphi \wedge \neg\psi) \equiv (\varphi \wedge \bigvee_i \neg c_i)$ is inconsistent iff $\varphi \wedge \neg c_i$ inconsistent for each c_i. Hence, this reduces to a number of CE checks.
  Therefore, **SE** is not supported efficiently.

- Bounded Conjunction ($\wedge$**BC**), General Conjunction ($\wedge$**C**)
  Given formulas $\varphi_1, \ldots, \varphi_n$, construct a formula representing $\varphi_1 \wedge \cdots \wedge \varphi_n$

  In order to construct the conjunction of n CNF formulas, we simply have to merge the clause sets. As this can be done by simply conjoining n formulas, $\wedge$**B** and $\wedge$**BC** are supported efficiently.

- Conjunction of Literals (**CL**)
  Given a conjunction φ of literals, construct a formula representing φ

  Each literal is a clause and hence their conjunction is a CNF formula.
  CL is supported efficiently.

- Renaming (**RN**), Renaming consistent with order ($\mathbf{RN}_{\prec}$)
  Given formula φ and an injective variable renaming $r : vars(\varphi) \to V'$, construct formula representing $\varphi[r]$ i.e. φ where each variable v is replaced by $r(v)$

We can rename any CNF formula in linear time in $|| \varphi ||$ by simply iterating over it and renaming each occurrence individually. In particular, we can rename in any order.
RN and is supported efficiently.

With exception to **CO**, **CE** and **SE**, all required queries are supported efficiently by CNF formulas.

5.2 Efficiency despite SAT

Although CNF formulas support many queries efficiently, they have an important drawback: they do not support **CO** and following from that also neither **CE** nor **SE**. Consistency is generally known as the *Boolean Satisfiability Problem* (SAT) and is prominently shown to be the first problem to be **NP**-complete (Cook, 1971).
For our purposes and under the assumption that $\mathbf{P} \neq \mathbf{NP}$, this means, that CNF formulas are not able to support consistency checks efficiently. Therefore, we will investigate whether propositional formulas in conjunctive normal form are a suitable formalism to represent inductive certificates, under the restriction to disregard their inefficiency regarding consistency checks.

6 Application to search algorithms

With the aim of generating plans for a planning task, planning systems commonly use a search mechanism to traverse the induced transition system. Graph search generally starts at one state and explores new states by applying actions to states seen previously. This is referred to as *expanding* states. The *open list* keeps track of which states should be expanded next and at each step in the search process, states are taken out in order to be expanded. Although not always necessary, some search algorithms also employ a *closed list* that stores already expanded states. In that case, states are only expanded if they are not in the closed list, which avoids expanding the same state multiple times.

The most prevalent search algorithms can be categorized in three families: *breadth-first*, *depth-first* and *best-first search*. Most notably, they differ in which order states in the open list are expanded. Breadth-first search expands states in the open list in LIFO order. Therefore, the search space is expanded layer by layer in the sense that at first states adjacent to the starting state are expanded. Then iteratively the states immediately adjacent to those states and so forth. Hence, breadth-first search explores the search space in a wave-formation. In contrast to that, depth-first search expands states in FIFO order, where most recently detected states are expanded first. This approach creates a more wide-spread and branched-out search-space.

Best-first search expands the most promising states in the open list according to a specific rule. For instance, uniform cost search assigns each state the cost of reaching it from the initial state. Hence, states that can be reached cheaply from the initial state are expanded first and since states are ordered by cost, this guarantees that once a solution is found, it is optimal. Commonly, best-fist search is implemented as *heuristic search*, where a *heuristic* is used as an estimate for the distance to the closest goal in combination with other information as the cost of reaching the state. This is the most prominent variant of best-first search, as it allows for a large variety of different flavors depending on the heuristic used.

Often, search algorithms make use of *pruning*, a powerful technique that helps to reduce the search space that must be explored. Common applications include the use of heuristics: under certain conditions, we can trust the judgment of a heuristic that a state will not lead to a goal state and simply omit to expand the state without missing out on potential solutions.

Generally, a planning system determines that a planning task is unsolvable once it has explored all states reachable from the starting point without ever reaching a goal state. Because pruning techniques potentially prevent the planning system to actually explore every single reachable state, this adds a layer of complexity to the process of proving unsolvability. We therefore have to look into methods that allow us to represent unexplored states fairly accurate.

In a first step, we will focus on *blind search* approaches which do not make use of pruning and examine how they can create certificates. Later on, we will augment this approach to be capable of representing unvisited states.

6.1 Blind Search

Blind search algorithms explore the search space by expanding states from the open list and stop when they reached a goal state. They are considered to be ”blind”, since they don’t make use of heuristics and pruning to speed up the search. More specifically, blind search is a best-fist search algorithm where the open list is ordered only according to the cost of reaching a state, without taking into consideration if an expanded state is actually useful for the search.
A progressive blind search starts at the initial state I^Π of a planning task Π. From there on, it generates and inserts all possible successor states using the applicable actions into the open list, in other words: it expands that state. The search continues until a goal state is expanded, or until all reachable states have been expanded and no goal state was reached. The set of all reachable states S from I^Π has three remarkable properties: it contains the initial state, it is inductive and if the task is unsolvable, it does not contain any goal states. Hence, it is an inductive certificate.
In the following, we look into how blind search can construct inductive certificate formulas.
Since blind search only expands a single state s at each iteration, we can easily generate a formula that describes all reachable states during the search as follows.
Initially, we start with the empty clause $\varphi = \bot$ and then simply update φ after expanding state s according to the update rule shown in Equation 6.1.

$$\varphi := \varphi \vee (\bigwedge_{v \in s} v \wedge \bigwedge_{v \notin s} \neg v) \tag{6.1}$$

Adding a new state to φ can be done in time linear in $\mid V^\Pi \mid$, since the formula added contains a literal for every variable in V^Π.
This procedure generates a formula φ by simply appending each visited state using a disjunction. The resulting formula represents all visited states, in the sense that each state visited is a model of φ. If S_R is the set of all reachable state, then $states(\varphi) = S_R$.
Notice that by disjoining visited states, φ is naturally generated as a disjunction of cubes. In fact, this is a normal form called disjunctive normal form and can be understood as the opposite to CNF in the sense that conjunctions and disjunctions are switched. Since modern SAT-solvers require their input to be in CNF we have to modify φ following one of two approaches: we could aim to transform φ from disjunctive into conjunctive normal form. However, this approach is only feasible if we allow introducing auxiliary variables during the transformation. We will revisit this approach in Chapter 7.2.1.
On the other hand, we could aim to generate a formula that is in CNF by construction. A natural way to achieve this focused on backwards inductive certificates. Backwards inductive certificates make use of the set of states which are not reachable by the initial state. In that sense, they can be interpreted as the dual of regular inductive certificate and can be used to derive formulas that represent state sets in conjunctive form immediately.
We can derive a formula φ_b representing the backwards inductive certificate similarly as before: initially, we start with the empty cube $\varphi_b = \top$ and update φ' according to Equation 6.2.

$$\varphi_b := \varphi_b \wedge (\bigvee_{v \in s} \neg v \vee \bigvee_{v \notin s} v) \tag{6.2}$$

Notice that φ_b is in conjunctive normal form by construction.
Further, we could obtain φ_b by generating φ first: Since φ describes all reachable states, its negation describes all states that are unreachable from the initial state I^{Π}. Since φ is a DNF formula we derive the CNF formula φ_b by simply negating φ.
Theorem 3 gives us that $\neg\varphi$ describes a backwards inductive certificate, and by Theorem 2 we have a sound and complete tool at our disposal.

6.2 Heuristic Search

Heuristics can play an important role in the process of finding a plan for a planning task. During search, the planning system makes use of information provided by a heuristic to guide the search towards a goal state. There is a huge variety of different heuristics that are used in state-of-the-art planning systems, such as relaxation heuristics, merge-and-shrink heuristics or heuristics based on landmarks. While heuristics usually are used to minimize search time, they serve a different purpose for the detection of unsolvable planning tasks. Heuristics work by associating a numerical value to a state that represents an estimate distance from that state to a goal state. If a heuristic determines that a goal is unreachable from a state, it is associated with an infinite heuristic value, which symbolizes that it is of no use to further expand the state. Therefore, assigning an infinite heuristic value corresponds to pruning the search tree. We call states with an infinite heuristic value *dead-end*.
In this chapter, we look into how inductive certificates can be generated for a specific class of heuristics. As blind search expands every reachable state, the set of expanded states corresponds to the set of reachable states. Naturally, the set of reachable states is inductive. If it does not contain any goal state it is an inductive certificate as the initial state must always be expanded during search.
In heuristic search, we often end up pruning states because they are classified as a dead-end by the heuristic. Although this accelerates the search process as we do not expand all reachable, we lose inductivity: not all reachable states are actually expanded, and hence the set of expanded states S_{exp} is not an inductive certificate.
Before we investigate how to obtain an inductive certificate when we make use of pruning, we introduce the following heuristic properties.

Definition 20 (safety and consistency)**.** *Let h be a heuristic and s a state.*
We call h safe *if it only assigns an infinite heuristic to state s if no goal is reachable from s.*
We call h consistent *if $h(s) \leq h(s[a]) + cost(a)$ for all states s and all applicable actions a, where $cost : A \rightarrow \mathbb{R}_0^+$ assigns each action a cost.*

For the following, we will assume that heuristic h is safe and consistent. As we see later on, with these properties we are able to obtain an inductive certificate by representing un-expanded states.
Let us at first consider the case, where only a single state s_d is declared a dead-end. If we assume, that we can represent the set R_{s_d} of all reachable states from s_d we can regain inductivity: since states in S_{exp} only lead to states in $S_{exp} \cup \{s_d\}$ and s_d only leads to states in R_{s_d}, the set $S_{exp} \cup R_{s_d}$ is inductive. An example is shown in Figure 6.1. Further, it always contains the initial state and because h safe, it can not contain a goal. Additionally, since h is consistent, no state reachable from s_d has a lower heuristic value and therefore, $S_{exp} \cup R_{s_d}$ is an inductive certificate.
Similarly, we can derive an inductive certificate formula: if we can represent the set R_{s_d} with formula φ_{s_d}, then the disjunction of φ_{s_d} and φ_{exp}, representing the set of expanded states, is an inductive certificate formula.

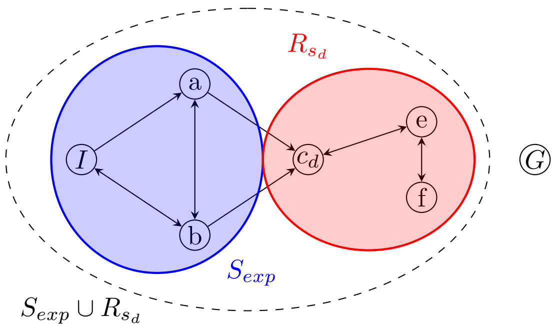


Figure 6.1: Example of search space in which a heuristic considers s_d as dead-end

However, in practice we won't always be able to represent R_{s_d} exactly, but rather make use of an inductive over-approximation. An inductive over-approximation is sufficient, because still S_{exp} only leads to states in itself or the inductive over-approximation and hence their union is inductive as well. In what follows, we will extend this approach to the case of multiple dead-end states. If a heuristic h is safe and consistent, then the union of all sets R_{s_d} for all dead-ends s_d is inductive and contains no goal state.

The following theorem shows the connection between these heuristic properties and the set of all dead-ends S_{inf}, that is, the states s with $h(s) = \infty$.

Theorem 4 (Eriksson, 2019a, Theorem 7.1)**.** *Given a heuristic h, let S_{inf} be the set of state $s \in S^{\Pi}$ with $h(s) = \infty$.*

a) If h is safe, S_{inf} contains no goal state.

b) If h is consistent, S_{inf} is inductive.

Theorem 4 provides the foundation for certifying unsolvability in the case of heuristic search: for a safe and consistent heuristic h, the set $S_{inf} \cup S_{exp}$ is an inductive certificate for Π. Therefore, if we can represent the set S_{inf} with a formula φ_{inf}, we acquire an inductive certificate formula.

Heuristics have certain criteria under which they label a state as dead-end. Although multiple heuristics are safe and consistent, we have to examine different heuristics separately and investigate how they can produce an inductive certificate formula.

In the following, we will look into how the family of delete relaxation heuristics can generate inductive certificates. Despite the fact that this heuristic family consists of multiple heuristics, we can analyze them as one, because they label a state as dead-end under the same condition.

6.2.1 Delete Relaxation

During search, it sometimes makes sense to look into an altered planning task in which the delete effects of all actions have been removed. This allows to interact with a simplified version of the planning task to generate information that can be used in the original planning task. Several different heuristics like $h^{+}, h^{max}, h^{add}, h^{FF}$ and h^{LM-cut}(e.g. (Bonet and Geffner, 2001), (Hoffmann and Nebel, 2001)) compute their values on this altered task and are summarized as the family of *delete relaxation heuristics*. The following definition formalizes the idea of removing delete effects of an action, which results in the delete relaxation of a planning task.

Definition 21 (delete relaxation)**.** *Let* $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$ *be a STRIPS planning task. The* delete relaxation *of* Π *is the planning task* $\Pi^R = \langle V^\Pi, A^R, I^\Pi, G^\Pi \rangle$ *where* A^R *contains an altered action* a^R *for each* $a \in A^\Pi$ *such that* $pre(a^R) = pre(a), add(a^R) = add(a)$ *and* $del(a^R) = \emptyset$*.*

Delete relaxations have an interesting and useful property: applying an arbitrary action in any state can never lead to a state with fewer variables, but only to a state with the same or more variables. This property arises immediately from the fact that the delete-effect of an action in a delete relaxation is always empty. Therefore, once a variable is true, it will be true for all of its successors. Although states in a delete relaxation can therefore be considered monotonic, this does not mean that it is necessarily possible to reach a state with all variables. Hence, it is reasonable to introduce the notion of unreachable variables.

Definition 22 (unreachable variable)**.** *Let* $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$ *be a STRIPS planning task,* $\Pi^R = \langle V^\Pi, A^R, I^\Pi, G^\Pi \rangle$ *its delete relaxation and s a state over the variables in* Π^R*. A variable* $v \in V^\Pi$ *is called* unreachable *in* Π^R *from state s, if it is impossible to reach a state* s' *from s that contains v. Further, we define the set of unreachable variables from state s as* $U^R_s := \{v \in V^\Pi \mid v \text{ is unreachable from } s\}$*.*

Since the states are monotonic in a delete relaxation, we can easily determine the set U^R_s for any state s with the following procedure:

1. Insert all variables of state s into V^R_s, the set of all reachable variables from s
2. Check if there is any unmarked, applicable action a^R to the set V^R_s. If there are none, return $U^R_s := V^\Pi \setminus V^R_s$
3. Take an action a^R that was found in step 2, insert the variables $add(a^R)$ into V^R_s, and mark a^R as applied
4. Jump to step 2

It is easy to see that V^R_s is the set of all reachable variables from state s, as variables are inserted into V^R_s only if there are reachable by some action sequence. From this set, we can simply derive the set of unreachable variables U^R_s as the set of all variables that are not contained in V^R_s.
Using the set U^R_s, we can now define an inductive set in the original task Π:

Theorem 5 (Eriksson, 2019a, Theorem 7.2)**.** *Given* U^R_s *for some state s, the set* $S^R_s = \{s' \subseteq V^\Pi \mid s' \cap U^R_s = \emptyset\}$ *is inductive in* Π*.*

Although delete relaxation heuristics can be quite different regarding heuristic guidance during search, they are identical when it comes to detecting dead-ends: a dead-end state s_d is detected as such by any delete relaxation if $U^R_{s_d}$ contains at least one goal variable. This is very natural as this simply means that at least one goal variable is unreachable from state s_d, hence there can not exist a plan that traverses s_d. If $U^R_{s_d}$ contains a goal variable, then the set $S^R_{s_d}$ contains no goal state. In particular, $S^R_{s_d}$ is an inductive over-approximation of the inductive set R_{s_d} containing all reachable states from dead-end s_d.
We can describe the set $S^R_{s_d}$ accurately by the formula φ_{s_d} in Equation 6.3.

$$\varphi_{s_d} = \bigwedge_{v \in U^R_{s_d}} \neg v \tag{6.3}$$

Additionally, we derive formula φ_{inf} representing the set S_{inf} as follows:

$$\varphi_{inf} = \bigvee_{s_d \text{ is a dead-end}} \varphi_{s_d}$$

Therefore, we can represent the set of all reachable states from I^Π for delete relaxation heuristics with $\varphi_{exp} \vee \varphi_{inf}$.
Whereas forward inductive certificates express the over-approximation of the set of reachable states from dead-end s_d with the set that does not contain any unreachable variable, backwards inductive certificate express this from a regression perspective: a state s is unreachable from dead-end s_d, if it contains any unreachable variable $U^R_{s_d}$ from s_d.
If we denote the set of unreachable states from s_d with $\overline{S^R_{s_d}}$, we can express this set with formula $\overline{\varphi_{s_d}}$ as depicted in Equation 6.4.

$$\overline{\varphi_{s_d}} = \bigvee_{v \in U^R_{s_d}} v \tag{6.4}$$

Similarly, we can derive a formula $\overline{\varphi_{inf}}$ representing the backwards inductive sets for each dead-end.

$$\overline{\varphi_{inf}} = \bigwedge_{s_d \text{ is a dead-end}} \overline{\varphi_{s_d}}$$

Notice that $\overline{\varphi_{s_d}} \equiv \neg\varphi_{s_d}$ and $\overline{\varphi_{inf}} \equiv \neg\varphi_{inf}$. Furthermore, $\overline{\varphi_{inf}}$ is in CNF by construction and φ_{s_d} only requires **CL**, which is supported efficiently by CNF.
Since a part of the delete relaxation heuristic computation already includes a relaxed reachability analysis, $U^R_{s_d}$ and therefore $S^R_{s_d}$ and $\overline{S^R_{s_d}}$ can be generated with no additional overhead. Theorem 6 captures these results.

Theorem 6. *Given a state s_d that is detected as dead-end by a delete relaxation heuristic. The formula φ_{s_d} representing the set $S^R_{s_d}$ can be constructed efficiently using **CL**. Further, we can generate φ_{inf} by using **CL** a number of times.*

This chapter showed specifically how to generate a formula that represents the set of reachable states in the case of blind search. For h^{max}, we are able to generate a formula that represents all expanded states and additionally all reachable states from each pruned state.

7 Validation Formulas

During search, the planner explores the state space with the aim of finding a plan for the underlying planning task. The planning system does not know in advance if a given planning task is solvable or not, and hence tries to determine if a plan exists. If the planning system finds a plan, it can be used as a certificate itself. In the event that the planning system comes to the conclusion that the given planning task is unsolvable, it returns a propositional formula as a certificate. This formula is an inductive validation formula representing the properties of an inductive or backwards inductive certificate according to Definition 15 and Definition 19 respectively.
In the following, we will investigate how such a formula, developed as a representative of the generated inductive certificate, can be constructed. We want to construct the formula such that we can validate it later with the help of a SAT-solver. Since SAT-solvers require a CNF formula as input, we will develop different approaches that enable us to use SAT-solvers for validation.

7.1 Generating Validation Formulas

An inductive certificate is given by a state set S, that satisfies the properties defined in Definition 15 or Definition 19. To construct a formula representing a forward or backwards inductive certificate, we have to consider their individual properties. As a reminder of these properties, an overview is given in Table 7.1.

forward	backward
$I \in S$	$I \notin \overline{S}$
$S \cap G \subseteq \emptyset$	$G \subseteq \overline{S}$
$S[A] \subseteq S$	$[A]\overline{S} \subseteq \overline{S}$

Table 7.1: Overview of properties of forward and backwards inductive certificates

Since we would like to represent the set-based properties as a propositional formula, we translate them with the help of Definition 17 into formulas.
In what follows, we write φ_X to denote a propositional formula that represents a state set X i.e. the models of φ_X represent states in X. Using the connectives $\wedge$ for $\cap$, $\neg$ for complement, we can express the composite set expression.

This is formalized with the use of an interpretation function I which maps set expressions to logical formulas.

$$
\begin{aligned}
I(X) &:= \varphi_X \text{ where } X \text{ is represented as } \varphi_X \\
I(\{I^\Pi\}) &:= \bigwedge_{v \in I^\Pi} v \wedge \bigwedge_{v' \notin I^\Pi} \neg v' \\
I(S^{G^\Pi}) &:= \bigwedge_{v \in g} v \\
I(\emptyset) &:= v \wedge \neg v \text{ for some } v \in V^\Pi \\
I(\overline{S}) &:= \neg I(S) \\
I(S \cap S') &:= I(S) \wedge I(S') \\
I(a) &:= \bigwedge_{v_p \in pre(a)} v_p \wedge \bigwedge_{v_a \in add(a)} v'_a \wedge \bigwedge_{v_d \in (del(a) \setminus add(a))} \neg v'_d \wedge \bigwedge_{v \in V^\Pi \setminus (add(a) \cup del(a))} (v \leftrightarrow v')
\end{aligned}
$$

In the following, we will use the formulas $\varphi_{I^\Pi}, \varphi_{G^\Pi}$ and φ_a to represent the corresponding interpretation functions.
Additionally, we denote the forward inductive formula with φ_S. Notice, that φ_S can be constructed as described in Chapter 6: for blind search, we append every reachable state to φ_S using Equation 6.1. In the case of delete relaxation heuristics, we additionally have to represent the set of pruned states for each dead-end. We do this by appending an over-approximation of the reachable states from a dead-end according to Equation 6.3.
With the help of these formulas, we can express the properties of forward inductive certificates as presented in Table 7.1.

- $I^\Pi \in S$
  check $\varphi_{I^\Pi} \models \varphi_S$ which is equivalent to checking that $\varphi_{I^\Pi} \wedge \neg\varphi_S$ unsatisfiable
  Since φ_{I^Π} is a simple conjunction of literals and in particular mentions every variable $v \in V^\Pi$, this check can be computed efficiently.

- $G^\Pi \cap S \subseteq \emptyset$
  check $\varphi_{G^\Pi} \wedge \varphi_S \models \varphi_\emptyset$ which is equivalent to testing that $\varphi_{G^\Pi} \wedge \varphi_S \wedge \neg\varphi_\emptyset$ unsatisfiable. Since the negation of the empty set formula evaluates to true, we can omit it. There we are left to check if $\varphi_{G^\Pi} \wedge \varphi_S$ is unsatisfiable

- $S[A^\Pi] \subseteq S$
  check $\forall a \in A^\Pi : \varphi_S \wedge \varphi_a \models \varphi_{S'}$ which is equivalent to $\varphi_S \wedge \varphi_a \wedge \neg\varphi_{S'}$ being unsatisfiable

In conclusion, we derive $2 + \#actions$ subformulas that need to be verified. Notice that because we want every subformula to be verified as unsatisfiable, we can construct a formula that represents all three properties by simply building the disjunction over all subformulas as follows:

$$
\varphi_S = (\varphi_{I^\Pi} \wedge \neg\varphi_S) \vee (\varphi_{G^\Pi} \wedge \varphi_S) \vee \left(\bigvee_{a \in A^\Pi} (\varphi_S \wedge \varphi_a \wedge \neg\varphi_{S'}) \right) \tag{7.1}
$$

It can easily be seen that Equation 7.1 is satisfiable if and only if at least one of the subformulas is satisfiable. Hence, it is unsatisfiable iff every subformula is unsatisfiable.
Therefore, Equation 7.1 can be used to validate any unsolvable planning task, as it is unsatisfiable iff the set of states S indeed is an inductive certificate according to Definition 15.

For that reason, we call the formula in Equation 7.2 a *forward inductive validation formula* and refer to it with φ_V

Notice however, that the validation formula is not in CNF and can therefore not yet be validated in a SAT-solver. We will investigate different approaches to validate the formula with a SAT-solver in Chapter 7.2.

Similarly to the forward inductive validation formula, we can derive a corresponding formula for backwards inductive certificate formulas.

For the following, the formula $\varphi_{\overline{S}}$ represents the set of all unreachable states $\overline{S}$, where $\varphi_{\overline{S}}$ was generated according to Equation 6.2.

- $I^\Pi \notin \overline{S}$
  check $\varphi_{I^\Pi} \not\models \varphi_{\overline{S}}$ which is equivalent to checking that $\varphi_{I^\Pi} \models \neg\varphi_{\overline{S}} \equiv \varphi_{I^\Pi} \wedge \varphi_{\overline{S}}$ is unsatisfiable
- $G^\Pi \subseteq \overline{S}$
  check $\varphi_{G^\Pi} \models \varphi_{\overline{S}}$ which is equivalent to $\varphi_{G^\Pi} \wedge \neg\varphi_{\overline{S}}$ being unsatisfiable
- $[A^\Pi]\overline{S} \subseteq \overline{S}$
  check $\forall a \in A^\Pi : \varphi_{\overline{S}'} \wedge \varphi_a \models \varphi_{\overline{S}}$ which corresponds to $\varphi_{\overline{S}'} \wedge \varphi_a \wedge \neg\varphi_{\overline{S}}$ being unsatisfiable

By constructing the disjunction of the subformulas we obtain a *backwards inductive validation formula* $\varphi_{\overline{V}}$ presented in Equation 7.2.

$$\varphi_{\overline{S}} = (\varphi_{I^\Pi} \wedge \varphi_{\overline{S}}) \vee (\varphi_{G^\Pi} \wedge \neg\varphi_{\overline{S}}) \vee \left(\bigvee_{a \in A^\Pi} (\neg\varphi_{\overline{S}} \wedge \varphi_a \wedge \varphi_{\overline{S}'}) \right) \tag{7.2}$$

Because of the identity $\neg\varphi_S \equiv \varphi_{\overline{S}}$, the inductive validation formulas presented in Equation 7.1 and Equation 7.2 are equivalent. Although it seemed to be useful at first to generate the formula φ_S representing the set of reachable states S as its dual in the form of $\varphi_{\overline{S}}$, the approaches turn out to be identical. This is consistent with our prior results, summarized in Theorem 3 and hence both of these formulas can be used as certificates for unsolvable planning tasks. For simplicity, we will only refer to one of the approaches, but notice that they can be used interchangeably.

7.2 Verifying Validation Formulas

In Chapter 7.1, we investigated how to derive an inductive validation formula that acts as a certificate for unsolvable planning tasks. We can construct the formula by expressing the properties of forward and backwards inductive certificates using propositional formulas for the set of reachable states, the initial state, the set of goal states and the set of actions given by a planning task. Constructing the disjunction of the resulting subformulas yields the inductive validation formula presented earlier in Equation 7.1.

$$\varphi_V := (\varphi_{I^\Pi} \wedge \neg\varphi_R) \vee (\varphi_{G^\Pi} \wedge \varphi_R) \vee \Big(\bigvee_{a \in A^\Pi} (\varphi_R \wedge \varphi_a \wedge \neg\varphi_{R'}) \Big) \tag{7.3}$$

Assume that a planning system comes to the conclusion that a presented planning task is unsolvable and returns a certificate in the form of a formula, as in Equation 7.3. Instead of trusting the judgement of the planning system, we are now able to validate whether the decision of the system was reached correctly or not.

As we have seen before, Equation 7.3 expresses the properties of an inductive certificate iff the formula is unsatisfiable. In that case, we know that the initial state is part of an inductive set that does not

contain any goal state. The existence of such a set immediately proves unsolvability of the planning task by Theorem 4.3 in Eriksson (2019a).
Therefore, we can validate the certificate, and with that the work of the certifying planning system, by testing the inductive validation formula φ_V for unsatisfiability in a SAT-solver of our choice.
However, φ_V is not in CNF, and hence we have to investigate how we can reformulate the formula before validating it. For this, we will consider two approaches:
the first approach transforms formula φ_V into a CNF formula. Here we will observe, that equivalent transformation cause an exponential blow-up and hence are not practical for our usage. Hence, we will look into a transformation algorithm that relaxes the guarantee of an equivalent transformation but still preserves all satisfying assignments. We will investigate the transformation approach more deeply in Chapter 7.2.1.
The second approach avoids transforming the φ_V by exploiting the structure of the formula. The idea is inspired by a simple observation: a formula $\psi = \bigvee_i \psi_i$, where ψ_i represents a subformula of ψ, is unsatisfiable iff every subformula ψ_i of ψ is unsatisfiable.
In the following, we will describe how we obtain much simpler subformulas from formula φ_V in Equation 7.3. For this, we will revisit the three properties of inductive certificates. Further, we will denote the forward inductive certificate formula representing the set of reachable states S constructed according to Equation 6.1 by φ_S. Remember that after visiting all reachable states, φ_S is of the following form: $\varphi_S = \bigvee_{s \in S}(\bigwedge_{v \in s} v \wedge \bigwedge_{v \notin s} \neg v)$.
The formula representing the initial and the goal states are denoted as φ_{I^Π} and φ_{G^Π} respectively, the formula representing the transition for an action a according to Equation 3.2 is referred to as φ_a and the formula of the empty set as $\varphi_\emptyset$.

- $I^\Pi \in S$
  $\varphi_{I^\Pi} \models \varphi_S$ which corresponds to $\varphi_{I^\Pi} \wedge \neg\varphi_S$ being unsatisfiable
  Notice that φ_S is in DNF, and that we can transform a negation of a DNF formula to CNF in linear time in the size of the formula by simply negating each literal and switching between $\wedge$ and $\vee$. Because φ_{I^Π} is a full assignment, SAT-solver can efficiently check for unsatisfiability by simple unit propagation.

- $G^\Pi \cap S \subseteq \emptyset$ $(\varphi_{G^\Pi} \wedge \varphi_S) \models \varphi_\emptyset$. This is equivalent to $\varphi_{G^\Pi} \wedge \varphi_S$ being unsatisfiable
  Because φ_S is a large disjunction over cubes, $\varphi_{G^\Pi} \wedge \varphi_S$ unsatisfiable iff $\varphi_{G^\Pi} \wedge \varphi_s$ unsatisfiable for all $s \in S$. Since φ_{G^Π} is in CNF already, we are left with the following validation checks:
  $\forall s \in S$: check $\varphi_{G^\Pi} \wedge \varphi_s$ unsatisfiable
  Notice, again, that φ_s mentions every variable and hence can be checked for unsatisfiability efficiently.

- $S[A^\Pi] \subseteq S_R$
  $\forall a \in A^\Pi : ((\varphi_S \wedge \varphi_a) \models \varphi_{S'})$ which is equivalent to checking that $\varphi_S \wedge \varphi_a \wedge \neg\varphi_{S'}$ is unsatisfiable
  Notice that φ_a is already in CNF because $(v \leftrightarrow v') \equiv ((v \vee \neg v') \wedge (\neg v \vee v'))$. Similar as before, $\neg\varphi_{S'}$ can be transformed to CNF in linear time, and by splitting up the calls for φ_S, we derive the following checks:
  $\forall a \in A^\Pi, \forall s \in S$: check $\varphi_s \wedge \varphi_a \wedge \neg\varphi_{S'}$ unsatisfiable
  This check can again be done efficiently, because this formula is a full assignment.

As seen above, we can split up the calls to the verifier and obtain much simpler formulas instead. The split allows us to completely avoid transforming the formula φ_V into CNF. This does not only safe a lot of effort in transformation time and memory, but also has a second important advantage: although we have to do many **CO** checks, they can be done efficiently because each formula is a full assignment.

7.2.1 Transforming the formula

In contrast to splitting up the formula and testing many subformulas for unsatisfiability, we can transform the formula in Equation 7.1 into a CNF formula. This leaves us with a single satisfiability check in CNF.
Transforming a propositional formula φ into CNF is commonly done with the use of De Morgan's law and the distributive property. In the following, v, w, q are propositional variables.

De Morgan's Law	Distributive property
$\neg(v \vee w) \iff (\neg v) \wedge (\neg w)$	$(v \wedge (w \vee q)) \iff ((v \wedge w) \vee (v \wedge q))$
$\neg(v \wedge w) \iff (\neg v) \vee (\neg w)$	$(v \vee (w \wedge q)) \iff ((v \vee w) \wedge (v \vee q))$

Table 7.2: Overview of De Morgan's law and distributive property

Let us consider formula φ in DNF shown in the Equation 7.4.

$$\varphi := (v_1 \wedge v_2 \wedge v_3 \wedge v_4) \vee (w_1 \wedge w_2 \wedge w_3 \wedge w_4) \tag{7.4}$$

If we would like to transform a formula in DNF into CNF, we have to apply the distributive property presented in Table 7.2 multiple times. This allows us to obtain the following equivalent formula in CNF.

$$\begin{aligned}
&(v_1 \vee w_1) \wedge (v_1 \vee w_2) \wedge (v_1 \vee w_3) \wedge (v_1 \vee w_4) \\
&\wedge (v_2 \vee w_1) \wedge (v_2 \vee w_2) \wedge (v_2 \vee w_3) \wedge (v_2 \vee w_4) \\
&\wedge (v_3 \vee w_1) \wedge (v_3 \vee w_2) \wedge (v_3 \vee w_3) \wedge (v_3 \vee w_4) \\
&\wedge (v_4 \vee w_1) \wedge (v_4 \vee w_2) \wedge (v_4 \vee w_3) \wedge (v_4 \vee w_4)
\end{aligned} \tag{7.5}$$

Constructing an equivalent CNF formula out of a DNF formula can be achieved easily, by simply building the disjunction over all variable pairs of the two subformulas. However, we transformed the compactly written formula φ to a much longer formula. Two formulas with four variables each are transformed into a formula with 16 clauses. In particular, Equation 7.5 has a lot more conjunctions than the number of disjunctions in φ. It is easy to see that in the case of n formulas with k variables each, a transformation yields k^n conjunctions. Unfortunately, this is not practical for our purposes, as we tend to deal with very large formulas.
But not all hope is lost: there exist efficient algorithms that can transform any formula into a CNF formula that is not equivalent to the original formula, but still preserves all satisfying assignments. This property is called *equisatisfiable*.

Definition 23 (equisatisfiable)**.** *Let φ and ψ be propositional formulas over a set of variables V. The formulas φ and ψ are* equisatisfiable *if φ is satisfiable iff ψ is satisfiable.*

Notice that the set of equivalent formulas is a subset of the set of equisatisfiable formulas: if two formulas are equivalent, they are also equisatisfiable, but not necessarily the other way around. The property of equisatisfiable is especially useful if we introduce auxiliary variables. If we know that we can preserve all satisfying assignment through a series of transformation steps, we can recover a model of the original formula by discarding the auxiliary variables.

Consider the case where we have two propositional formulas φ and ψ over the variable set $V \cup V^A$, where $V = \{v, w\}$ is a set of propositional variables and $V^A = \{x\}$ is a set of auxiliary variables. Further, assume that we have introduced auxiliary variables in only one of them, let's say in ψ, and still have the property of φ and ψ being equisatisfiable. Table 7.3 shows the equisatisfiable formulas φ and ψ with a list of their satisfying assignments.

$\varphi := (v \vee w)$	$\psi := (x \wedge (\neg x \vee v \vee w) \wedge (\neg v \vee x) \wedge (\neg w \vee x))$
$\{v \mapsto T, w \mapsto F\}$	$\{v \mapsto T, w \mapsto F, x \mapsto T\}$
$\{v \mapsto F, w \mapsto T\}$	$\{v \mapsto F, w \mapsto T, x \mapsto T\}$
$\{v \mapsto T, w \mapsto T\}$	$\{v \mapsto T, w \mapsto T, x \mapsto T\}$

Table 7.3: Equisatisfiable formulas φ and ψ with their models

However, notice that φ and ψ are not equivalent, as can be seen by the following counterexample: the assignment $\{v \mapsto T, w \mapsto F, x \mapsto F\}$ is a model for φ as the assignment to the unmentioned auxiliary variable x can simply be ignored. However, it is not a satisfying assignment for ψ, as can be seen in Table 7.3.

7.2.2 Tseitin Transformation

Satisfiability preserving transformations form the basis of polynomial translation algorithms from any propositional into a CNF formula, such as the *Tseitin transformation* (Tseitin, 1983). The Tseitin transformation is a tool to transform an arbitrary combinatorial logic circuit into an equisatisfiable propositional formula in conjunctive normal form. Although it is possible to transform any logic circuit into an equisatisfiable CNF formula, we will restrict ourselves to propositional formulas. Propositional formulas are a special case of a logic circuits as circuits may have multiple outputs, whereas formulas only have one output: if interpreted, they either evaluate to *true* or *false*.
The Tseitin transformation, also referred to as *Tseitin Encoding*, introduces a new variable x_i for each subformula φ_i with the idea, that the newly introduced variable represents the value of the corresponding subformula. Depending on the structure of subformula φ_i, we can make use of different pre-calculated CNF formulas to represent a 1-to-1 correspondence between φ_i and x_i. We can express this correspondence as a propositional formula as follows: $x_i \leftrightarrow \varphi_i$. By simply building the conjunction of all pre-calculated CNF formulas together with the substitute variable of the entire formula, we obtain an equisatisfiable formula φ_{T-CNF} in CNF. Since all pre-calculated CNF formulas are of fixed size, the resulting formula φ_{T-CNF} only grows linearly in the size of the original formula φ. This makes it a very useful tool, as it avoids the exponential blow-up described above, and hence can be used for an efficient transformation from DNF to an equisatisfiable CNF.
The following equation describes how a disjunction of two proportional variables v and w are transformed into CNF while introducing a new variable x representing the conjunction.

$$\begin{aligned} x \leftrightarrow (v \vee w) &\equiv (x \rightarrow (v \vee w)) \wedge ((v \vee w) \rightarrow x) \\ &\equiv (\neg x \vee v \vee w) \wedge (\neg(v \vee w) \vee x) \\ &\equiv (\neg x \vee v \vee w) \wedge ((\neg v \wedge \neg w) \vee x) \\ &\equiv (\neg x \vee v \vee w) \wedge (\neg v \vee x) \wedge (\neg w \vee x) \end{aligned} \tag{7.6}$$

Similarly, we can derive expressions for other common subformulas. Some examples are presented in Table 7.4.

Subformula	CNF Equivalent
$x \leftrightarrow (v \vee w)$	$(\neg x \vee v \vee w) \wedge (\neg v \vee x) \wedge (\neg w \vee x)$
$x \leftrightarrow \neg(v \vee w)$	$(x \vee v \vee w) \wedge (\neg v \vee \neg x) \wedge (\neg w \vee \neg x)$
$x \leftrightarrow (v \wedge w)$	$(x \vee \neg v \vee \neg w) \wedge (v \vee \neg x) \wedge (w \vee \neg x)$
$x \leftrightarrow \neg(v \wedge w)$	$(\neg x \vee \neg v \vee \neg w) \wedge (v \vee x) \wedge (w \vee x)$
$x \leftrightarrow \neg v$	$(\neg v \vee \neg x) \wedge (v \vee x)$

Table 7.4: Common subformulas and their equivalent CNF formula

Following the derivation in Equation 7.6, we can extend the substitution rules to formulas with three, as can be seen in Equation 7.7.

$$x \leftrightarrow (v \vee w \vee q) \equiv (\neg x \vee v \vee w \vee q) \wedge (\neg v \vee x) \wedge (\neg w \vee x) \wedge (\neg q \vee x) \tag{7.7}$$

In particular, we easily obtain pre-calculated subformulas for every fixed number of variables n.

$$x \leftrightarrow \bigvee_{i=1}^{n} v_i \equiv \left(\neg x \vee \bigvee_{i=1}^{n} v_i\right) \wedge \left(\bigwedge_{i=1}^{n} (\neg v_i \vee x)\right) \tag{7.8}$$

Similarly, we can derive expressions for differently structures subformulas, as seen in Table 7.4. These substitution formulas, as presented in Equation 7.8, allow transforming formulas with large disjunction into CNF very compactly. A disjunction of n variables can be transformed into a CNF formula with $n + 2$ conjunctions, where one clause mentions all variables, n clauses mention only two variables and the final clause only consists out of the substitution variable x. Notice that the final clause is very important, since only that clause actually represents that the original formula must hold. All other clauses simply represent a substitution of a variable for a formula. For instance, consider Equation 7.6. The formula only represents that variable x is true iff $(v \vee w)$ is true, but does not represent the formula $(v \vee w)$ itself.

With the substitution rules presented in Table 7.4 and their respective generalizations to larger subformulas, we can transform a large variety of formulas already. As an example, we will revisit Equation 7.4.

$$\varphi := (v_1 \wedge v_2 \wedge v_3 \wedge v_4) \vee (w_1 \wedge w_2 \wedge w_3 \wedge w_4)$$

In a first step, we will introduce a new variable for each subformula. Notice, that we can make use of the generalized substitution formulas as φ represent a disjunction of two formulas with four variables each.

$$x_1 \leftrightarrow (v_1 \wedge v_2 \wedge v_3 \wedge v_4)$$
$$x_2 \leftrightarrow (w_1 \wedge w_2 \wedge w_3 \wedge w_4)$$
$$x_3 \leftrightarrow (x_1 \vee x_2)$$

In this case, x_3 corresponds to the entire formula φ. By simply building the conjunction of all these substitutions and the substitution variable for φ, we obtain a transformed and equisatisfiable version of the original formula:

$$(x_1 \leftrightarrow (v_1 \wedge v_2 \wedge v_3 \wedge v_4)) \wedge (x_2 \leftrightarrow (w_1 \wedge w_2 \wedge w_3 \wedge w_4)) \wedge (x_3 \leftrightarrow (x_1 \vee x_2)) \wedge x_3$$

Each substitution can now be transformed into CNF using pre-calculated subformulas resulting in the formula shown in Equation 7.9.

$$
\begin{aligned}
&(\neg x_1 \vee v_1 \vee v_2 \vee v_3 \vee v_4) \wedge (x_1 \vee \neg v_1) \wedge (x_1 \vee \neg v_2) \wedge (x_1 \vee \neg v_3) \wedge (x_1 \vee \neg v_4) \\
&\wedge(\neg x_1 \vee w_1 \vee w_2 \vee w_3 \vee w_4) \wedge (x_1 \vee \neg w_1) \wedge (x_1 \vee \neg w_2) \wedge (x_1 \vee \neg w_3) \wedge (x_1 \vee \neg w_4) \\
&\wedge(\neg x_3 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge x_3
\end{aligned} \tag{7.9}
$$

Notice, that the generalized substitution rules allow us to transform large disjunctions/conjunctions very efficiently. We only introduce a single auxiliary variable to present the disjunction/conjunction, whereas the simple Tseitin transformation would introduce additional auxiliary variables for each pair of literals.

In summary, we derived two different CNF formulas from the original formula φ presented in Equation 7.4. Transforming φ into an equivalent CNF formula resulted in an impractical blow up of clauses. Therefore, we utilized the Tseitin transformation to obtain an equisatisfiable formula in CNF by introducing auxiliary variables. This transformation preserves satisfiability of the original formula and increases the formula size only linearly.

A comparison between the two transformations regarding the number of clauses in the resulting CNF formula can be seen in Figure 7.1. In particular, we distinguish between the simple and generalized Tseitin transformation.

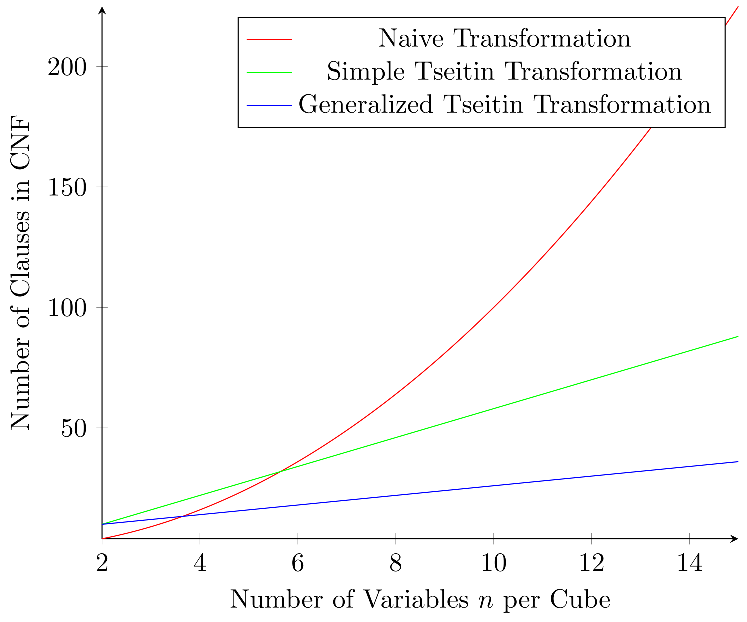


Figure 7.1: Number of Clauses after transforming $(\bigwedge_{i=1}^{n} v_i) \vee (\bigwedge_{i=1}^{n} w_i)$ into CNF

The Tseitin transformation is a powerful algorithm that allows to transform arbitrary propositional formulas into an equisatisfiable CNF formulas. In particular, it offers a way to translate the inductive validation formula φ_V into a suitable form for a SAT-solver. As we only care about if the validation formula is satisfiable and do not actually look for a satisfying assignment, the Tseitin transformation is a very useful tool for our purposes.

8 Experimental Setup

To evaluate the methods presented in this work from a practical standpoint, we extended the Fast Downward planning system (Helmert, 2006) to be able to generate backwards inductive certificates formulated as CNF formulas. For the scope of this thesis, we restricted ourselves to A* search combined with blind search and the delete relaxation h^{max}. We used three separate approaches to generate the final CNF formula: the first approach avoids any transformation of the formula into CNF. Instead, it checks for unsatisfiability of the inductive validation formula by testing each subformula independently. In the second approach, we focused on minimal overhead in the Fast Downward planning system, by generating certificates as boolean circuits and using the transformation tool `bc2cnf` introduced by Junttila and Niemelä (2000). The translation is based on the Tseitin transformation and utilized to transform the generated propositional formula into conjunctive normal form. This transformation tool allows us to convert boolean circuits into CNF using simple Tseitin transformations. In the third approach, we implemented a version of the Tseitin transformation directly into Fast Downward. This allows to avoid reading and writing potentially large files and exploit the structure of the formulas generated by the planning system. Further, this approach does not translate a formula, but immediately writes a CNF.

Independent of how the CNF formulas are generated, in a last step, they can be validated using state-of-the-art SAT-solvers. For the transformation based approaches, we used the `Kissat` SAT-solver and for the approach breaks the formula into more practical pieces we use `CaDiCal`, both developed by Biere, Fazekas, et al. (2020). For the evaluation, we used the experiment benchmark assembled and described in more detail in Eriksson (2019a) consisting exclusively of unsolvable planning problems. The benchmark is publicly available (Eriksson, 2019b). Further, we utilized the Downward Lab toolkit (Seipp et al., 2017) as a framework for running the experiments.

The experiments are run on a cluster consisting of Intel Xeon E5-2660 2.2 GHz processors. The planning system was given 3584 MiB memory and 30 minutes time. We used the same memory limit for the translation tool `bc2cnf` and for validating the certificates with the SAT-solvers, but adjusted the time limit to 60 minutes for `bc2cnf` and 4 hours for the SAT-solvers. Although this would mean, that the `bc2cnf` approach has technically 60 minutes more time, `bc2cnf` did not once fail because the time restriction. In fact, it never needed more than two minutes transformation time, but is much more restricted by the memory limitation. The three steps generation, transformation and verification all use a single CPU core.

8.1 SAT-Solver

To validate the certificates generated through one of the two transformation based approaches, we use a SAT-solver called `Kissat` (Biere, Fazekas, et al., 2020). `Kissat` is not only the main track winner of the 2020 SAT competition, but also placed first on a different branch focusing on unsatisfiable instances. For our experiments, we will use the 2020 version that was submitted to the SAT competition 2020[2], as it has a proven record in working with unsatisfiable formulas. The solver has an option to target such formulas, and we will make use of that option for the experiments. Otherwise, we will run `Kissat` on default configurations.

To validate the formulas that are part of the split-up approach, we use `CaDiCal`, another SAT-solver by Biere, Fazekas, et al. (2020). We decided against the use of `Kissat` for this, because it currently does not fully support the incremental interface `IPASIR`[3]. This interface provides an API to the SAT-solver and additionally provides useful functionalities. In particular, it allows to solve a series of similar formulas efficiently by remembering parts of the formula. Since the split-up approach would otherwise require us to read-in a potentially long formula countless times, it is only reasonable to utilize the provided interface.

Modern SAT-solver require a specific input format called *DIMACS CNF*. This format is a specific textual representation of a CNF formula and will be mentioned a few times in Chapter 9.

[2] https://satcompetition.github.io/2020/
[3] https://github.com/biotomas/ipasir

9 Experimental Results

We tested the generation of inductive validation formulas and the following verification through the SAT-solvers `Kissat` and `CaDiCal` on the benchmark mentioned in Chapter 8. For this, we ran the following configurations in Fast Downward:

- *blind*: A* search guided by a blind heuristic. States are expanded according to their cost from the inital state.
- h^{max}: A* search guided by the h^{max} heuristic. States are expanded according to their heuristic value and the cost of reaching the state.

To analyze the overhead caused by the generation of the inductive validation formulas, we also ran an unaltered version of Fast Downward and denote it in the following by FD. The first version that certifies unsatisfiability by splitting the formula into simpler CNF formulas is denoted by FD^{Sp}, the second approach utilizing `bc2cnf` is referred to as FD^{BC} and its transformation algorithm is denoted by Trans^{BC}. Finally, FD^{D} stands for the third approach that makes use of an implemented Tseitin transformation. The corresponding verifiers are indicated by VER^{Sp} , VER^{BC} and VER^{D}. Overall, all three approaches have certain characteristics that make them advantageous from a specific perspective in comparison to the others.
In the following sections, we will describe parts of the implementation, and examine the results of the generation and validation of the certificates and finally summarize the results.

9.1 Implementation

If the augmented version of Fast Downward comes to the conclusion that a presented planning task is unsolvable, it must create a forward or backwards inductive validation formula. We decided to implement the certificate from a backwards inductive view. The backwards inductive validation formula consists out of the following subformulas:

- formulas $\varphi_{\overline{R}}$ and $\varphi_{\overline{R}'}$ representing the set of unreachable states and their successors
- formula $\varphi_{I^{\Pi}}$ representing the initial state
- formula $\varphi_{G^{\Pi}}$ representing the goal states
- formula φ_a representing the transition induced by each action a

Instead of validating the validation formula in Equation 7.2 after a CNF transformation in one step, $\mathrm{FD}^{\mathrm{Sp}}$ shows unsatisfiability of each subclause as shown in Chapter 7.2. More specifically, $\mathrm{FD}^{\mathrm{Sp}}$ does not generate the validation formula itself, but rather a compact task description and the backwards inductive certificate formula. In a second step, a targeted verifier $\mathrm{VER}^{\mathrm{Sp}}$ produces subformulas that are validated immediately.
Notice that $\mathrm{FD}^{\mathrm{BC}}$ uses `bc2cnf` to transform the formula into DIMACS CNF format. This permits $\mathrm{FD}^{\mathrm{BC}}$ to generate a relatively compact formula. Since `bc2cnf` requires Boolean Circuits as input, we write each subformula as a gate and later on generate the inductive validation formula by simply using the connectives "$\vee$" and "$\wedge$". The generation process is over after we have written all formulas in the required style.
FD^{D} avoids writing a temporary file to be translated into DIMACS CNF, by simply transforming the formula immediately. Here, we utilized the generalized substitution formulas presented in Equation 7.8, where we introduced an auxiliary variable for each state, each property and the final formula as is described in the following. Here we will denote the union of the set of expanded states and the set of dead-end formulas with S_R.

- $x_i \leftrightarrow (\bigvee_{v \in s_i} \neg v \vee \bigvee_{v \notin s_i} v)$ for each state $s_i \in S_R$
- $x_{\overline{R}} \leftrightarrow \bigwedge_{s_i \in S_{\overline{R}}} x_i$
- $x_{I^\Pi} \leftrightarrow (\bigwedge_{v \in I^\Pi} v \wedge \bigwedge_{v' \notin I^\Pi} \neg v')$
- $x_{init} \leftrightarrow (x_{\overline{R}} \wedge x_{I^\Pi})$
- $x_{G^\Pi} \leftrightarrow \bigwedge_{v \in G^\Pi} v$
- $x_{goal} \leftrightarrow (\neg x_{\overline{R}} \wedge x_{G^\Pi})$
- $x'_i \leftrightarrow (\bigvee_{v \in s'_i} \neg v \vee \bigvee_{v \notin s'_i} v)$ for each state $s'_i \in S'_R$
- $x_{\overline{R}'} \leftrightarrow \bigwedge_i x'_i$
- for each action a:
  $x_a \leftrightarrow (\bigwedge_{v_p \in pre(a)} v_p \wedge \bigwedge_{v_a \in add(a)} v'_a \wedge \bigwedge_{v_d \in (del(a) \setminus add(a))} \neg v'_d \wedge \bigwedge_{v \in V^\Pi \setminus (add(a) \cup del(a))} (v \leftrightarrow v'))$
- $x_{actions} \leftrightarrow \bigvee_{a \in A^\Pi} x_a$
- $x_{ind} \leftrightarrow (\neg x_{\overline{R}} \wedge x_{\overline{R}'} \wedge x_{actions})$
- $x_V \leftrightarrow (x_{init} \vee x_{goal} \vee x_{ind})$
- x_V

Notice that we can easily write the final equijunction in x_a as a CNF formula by using $(v \leftrightarrow v') \equiv (v \vee \neg v') \wedge (\neg v \vee v')$. The resulting substitution step was omitted here for sake of clarity.
The last formula in the list x_V has an important role. Only that literal actually represents the backwards inductive validation formula. Without it, we would have only introduced auxiliary variables for subformulas, but not a validation formula.
The unsolvability certificate is only generated once the planning system comes to the conclusion that the task is unsolvable. If we begin writing the formula during search, we would introduce overhead in the case of a solvable task. In that case, the planning system obviously would not need the certificate.

After the search is concluded, we iterate over all visited states and begin to generate the formulas described above.

In the case of blind search, each visited state has a direct impact on the validation formula: we append the negation of the visited state to the formula representing the unreachable states. Because blind search does not use any pruning techniques, this approach is valid and produces a formula that accurately represents the set of unreachable states.

For h^{max}, this is different, since it makes use of pruning by determining that a state is a dead-end. In that case, the state is not expanded and hence its successors are not considered in the inductive validation formula. Therefore, if we encounter a dead-end while iterating over the visited states, we determine the set of unreachable variable with the procedure described in Chapter 6. Notice that to avoid overhead during the actual search, we don't save the unreachable variables that have been found in the calculation of the heuristic values. Instead, we recalculate them after the search terminated, which on one hand allows for a memory efficient search, but on the other hand requires additional computation after its termination.

The conjunction of the negated unreachable variables, as presented in Equation 6.3, can then be used to describe the dead-end state in the formula. For backwards inductive certificate formulas, this expression is simply negated before appending it.

	FD	FD^{Sp}	VER^{Sp}	FD^{BC}	Trans^{BC}	VER^{BC}	FD^{D}	VER^{D}
3unsat(30)	15	15	5	15	5	5	10	10
bag-barman(20)	12	12	0	8	0	0	8	0
bag-gripper(25)	3	3	0	3	0	0	2	2
bag-transport(29)	7	7	1	6	2	1	6	3
bottleneck(25)	10	10	4	9	5	4	8	6
cave-diving(25)	7	7	2	7	4	4	7	5
chessboard-pebbling(23)	5	5	2	5	2	2	4	3
diagnosis(13)	4	4	1	4	1	1	2	2
document-transfer(20)	5	5	1	5	3	2	5	3
mystery(9)	2	2	0	1	0	0	1	0
nomystery(150+24)	34	34	0	18	0	0	16	0
pegsol(24)	24	24	14	24	16	16	24	19
pegsol-row5(15)	5	5	1	5	3	3	4	4
rovers(150+20)	10	10	0	8	0	0	6	0
sliding-tiles(20)	10	10	0	10	0	0	10	0
tetris(20)	10	10	5	5	5	5	5	5
tpp(30+25)	24	24	3	17	7	6	15	9
total(697)	187	187	39	150	53	49	133	81

Table 9.1: Coverage overview for blind search

9.2 Generation

The three certifying planning systems FD^{Sp}, FD^{BC} and FD^{D} have overall great success in the generation of certificates. Table 9.1 and Table 9.2 present on overview of the coverage of all three approaches. For both blind search and h^{max}, FD^{Sp} is most successful. FD^{Sp} is able to generate a certificate for all tasks where FD is able to determine the task as unsolvable for blind search, whereas FD^{BC} and FD^{D} manage to produce a certificate in 80% and 71% respectively. For h^{max} we observe very similar results, as FD^{Sp}, FD^{BC} and FD^{D} are able to generate a certificate in 83%, 73% and 72% of cases respectively.

	FD	FD$^{\mathrm{Sp}}$	VER$^{\mathrm{Sp}}$	FD$^{\mathrm{BC}}$	Trans$^{\mathrm{BC}}$	VER$^{\mathrm{BC}}$	FD$^{\mathrm{D}}$	VER$^{\mathrm{D}}$
3unsat(30)	15	10	5	10	10	5	10	10
bag-barman(20)	8	8	0	8	0	0	8	0
bag-gripper(25)	2	2	0	2	0	0	2	2
bag-transport(29)	6	6	1	6	2	1	6	3
bottleneck(25)	21	17	10	16	12	11	16	14
cave-diving(25)	7	7	2	7	5	5	7	5
chessboard-pebbling(23)	5	5	2	5	2	2	4	3
diagnosis(13)	5	5	4	5	4	4	5	4
document-transfer(20)	7	7	3	6	5	4	6	5
mystery(9)	2	2	0	1	0	0	1	0
nomystery(150+24)	59	41	1	27	4	4	27	12
pegsol(24)	24	24	12	24	18	16	24	20
pegsol-row5(15)	5	5	2	5	3	3	4	4
rovers(150+20)	15	15	0	9	3	3	9	7
sliding-tiles(20)	10	10	0	10	0	0	10	10
tetris(20)	5	5	5	5	5	5	5	5
tpp(30+25)	23	23	6	14	7	7	14	9
total(697)	219	182	54	160	80	70	158	113

Table 9.2: Coverage overview for h^{max}

	blind		h^{max}	
	memory	time	memory	time
FD$^{\mathrm{Sp}}$	0	0	5	32
FD$^{\mathrm{BC}}$	0	37	25	34
Trans$^{\mathrm{BC}}$	96	1	77	3
FD$^{\mathrm{D}}$	1	53	21	40

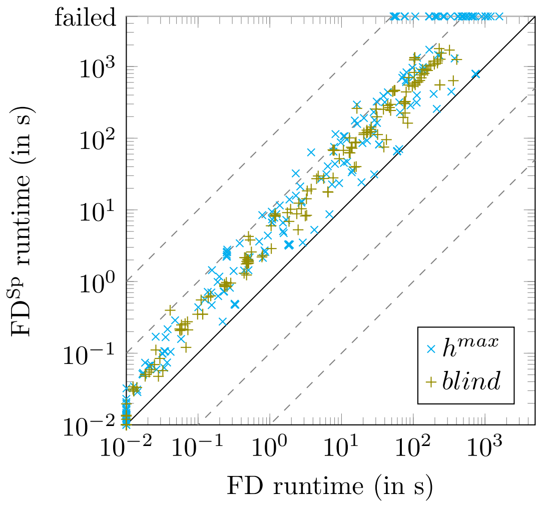
Table 9.3: Reason for failures during generation in tasks where FD generated a certificate

However, we need to be a bit more careful: FD$^{\mathrm{BC}}$ only has a valid certificate after Trans$^{\mathrm{BC}}$ transformed the received boolean circuit into DIMACS CNF, and hence we won't compare the output of FD$^{\mathrm{BC}}$ to the other two approaches. The results a very different if we consider Trans$^{\mathrm{BC}}$ instead, which was able to successfully generate a certificate in 28% for blind search and 36% for h^{max} of the cases where FD was able to determine the planning task as unsolvable.

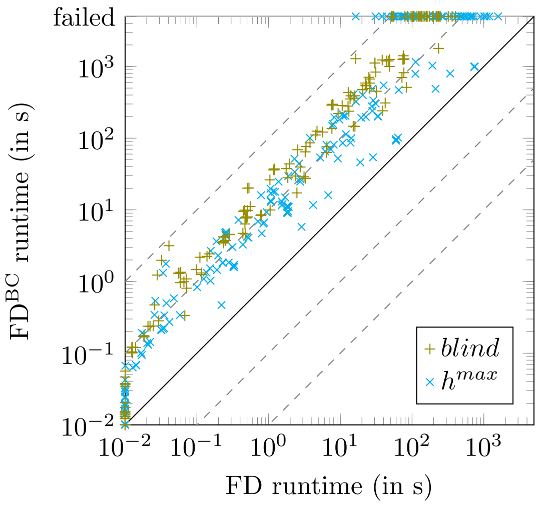
Because of extensive file writing, both FD$^{\mathrm{D}}$ and FD$^{\mathrm{BC}}$ mainly fail due to time, which can be seen in Table 9.3. In both cases h^{max} fails much more often to memory than blind search, which can be explained by the additional computation needed. Furthermore, FD$^{\mathrm{Sp}}$ does not fail once for blind search and mostly for time in the case of h^{max}. Since writing to the file-system is often time-consuming, timeouts are probably due to extensive writing.

However, notice that Trans$^{\mathrm{BC}}$ fails almost exclusively due to memory, as Trans$^{\mathrm{BC}}$ is based on the simple Tseitin transformation, which introduces an auxiliary variable for each subformula. Because of that, the simple transformation is relatively memory inefficient in comparison with the generalized transformation employed in FD$^{\mathrm{D}}$ as hinted towards in Figure 7.1.

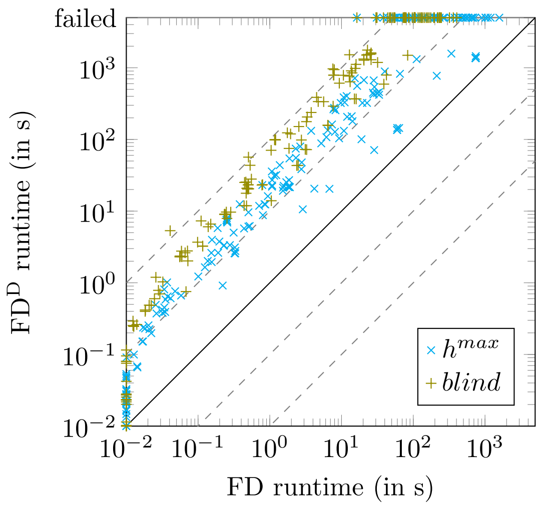
When it comes to the overall time consumption of the three certifying algorithms, we observe a similar pattern. In all cases, we have that FD$^{\mathrm{Sp}}$ is the fastest, FD$^{\mathrm{BC}}$ comes second and FD$^{\mathrm{D}}$ is the slowest as can be seen in Figure 9.1a, Figure 9.1b and Figure 9.1c. Additionally, we have to consider that the transformation in Trans$^{\mathrm{BC}}$ requires significant time as well. Typically, the transformation takes about $4 - 6$ as longer than FD$^{\mathrm{BC}}$ itself. Notice, however, that all considered methods impose moderate to considerable overhead to the planning system.



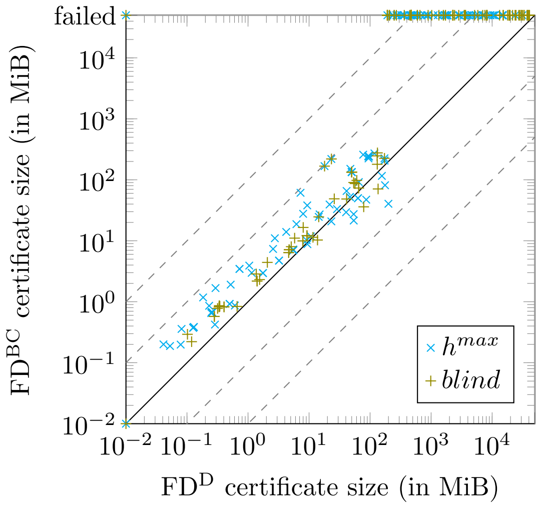
(a) Generation time FD$^{\text{Sp}}$



(b) Generation time FD$^{\text{BC}}$



(c) Generation timeFD$^{\text{D}}$



(d) certificate size of FD$^{\text{D}}$vs FD$^{\text{BC}}$

Figure 9.1: Overview of Generation time of FD$^{\text{Sp}}$ FD$^{\text{D}}$ and FD$^{\text{BC}}$ and comparison of certificate size for FD$^{\text{D}}$ and FD$^{\text{BC}}$

As writing the file is comparatively time-consuming, the results are understandable. FD$^{\text{Sp}}$ only generates two small files containing a task description and a backwards inductive certificate formula, whereas FD$^{\text{BC}}$ and FD$^{\text{D}}$ both produce a representation of the validation formula, which is typically much larger as it contains information about each action, the initial and goal states. In particular, FD$^{\text{BC}}$ produces a relatively compact description of the validation formula to transformed using Trans$^{\text{BC}}$. On the other hand, FD$^{\text{D}}$ produces a valid DIMACS CNF already, and is hence expected to be larger than the other files.

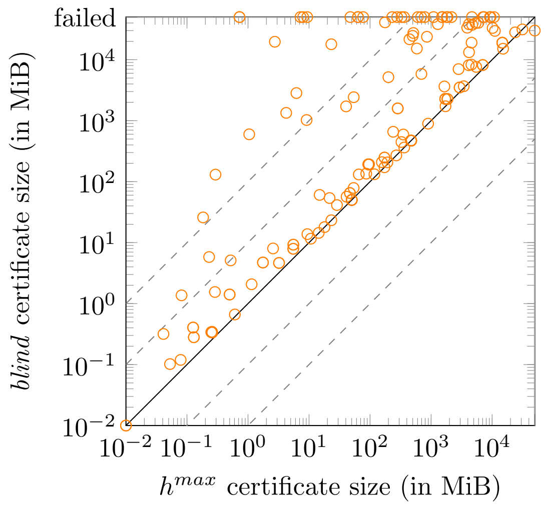
Whereas FD$^{\text{D}}$ produces certificates with an average size of 4GiB ranging up to around 20GiB and the certificates generated by FD$^{\text{Sp}}$ have a similar size with an average of 3.7GiB, this is very different for Trans$^{\text{BC}}$. Trans$^{\text{BC}}$ produces certificates with an average size of 57MiB. One reason for this is the high number of memory failures of Trans$^{\text{BC}}$ presented in Table 9.3, because Trans$^{\text{BC}}$ simply fails for instances that are too large to transform. On the other hand, FD$^{\text{D}}$ and FD$^{\text{Sp}}$ are able to generate much larger certificates. without failing.

However, as mentioned before, it makes sense to compare the output of Trans$^{\text{BC}}$ instead of the output of FD$^{\text{BC}}$. A comparison of the respective DIMACS CNF can be found in Figure 9.1d. We can see that FD$^{\text{D}}$ is able to generate certificates that are in most cases smaller than the certificates produced by Trans$^{\text{BC}}$.

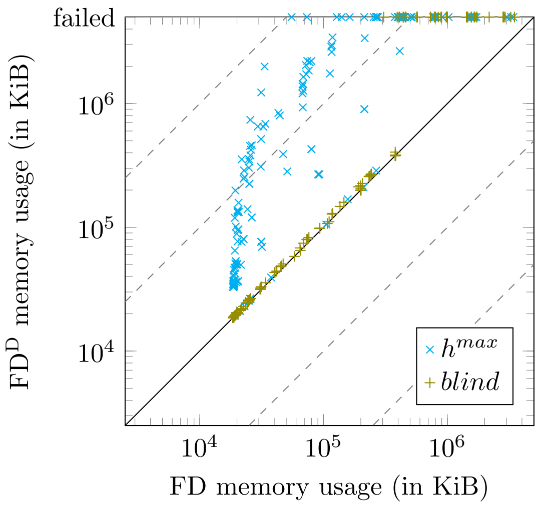
The memory consumption of all three approach is for the most part identical. In total, we observe deviations between FD$^{\text{Sp}}$, FD$^{\text{BC}}$ and FD$^{\text{D}}$ of less than 0.01%. This is only natural, since all approaches are based on the same search algorithm and none of them needs to store much exclusive information.

Furthermore, there are interesting observations when comparing blind search to h^{max}. Almost always, the certificates of h^{max} are smaller than for blind search due to the fact that the detection of dead-ends allows us to prune the search tree, as is presented in Figure 9.2a. Dead-ends permit space reduction in two ways: by pruning the search space at a dead-end we decrease the set of expanded states, which, depending on the number of dead-ends determined, reduces the number of explicitly written states in the certificate drastically. In most cases, h^{max} was able to reduce the number of states that need to be expanded. This reduced the number of expansions, sometimes only marginally and ranging up to a factor of multiple thousands. Only if h^{max} does not report any dead-ends, both approaches expanded all reachable states. In that case, the resulting certificate files trivially have the same size, as from an unsolvability perspective, h^{max} behaves identical to blind search.

Additionally, although not nearly as impactful, if h^{max} encounters a dead-end, we need to represent the inductive set $S^R_{s_d}$ of reachable states from the dead-end instead of the state itself. For h^{max} we use an over-approximation of the set $S^R_{s_d}$ presented in Equation 6.3. This expression is based on the set of unreachable variables and can be represented compactly. Therefore, even if a dead-end might not have any successors, we save space in comparison to blind search.



(a) time



(b) memory

Figure 9.2: Certificate size comparison of blind search and h^{max}. Memory of FD$^{\text{D}}$

However, the space reduction comes at a price: the heuristic h^{max} requires additional computation to determine the set of unreachable variables for each dead-end. A comparison of the memory consumption of FD$^{\text{D}}$ and FD is shown in Figure 9.2b. Notice that because of the identical memory consumption of the three approaches, we chose to show this for only one of them. The large memory deviation between blind search and h^{max} is due to the fact that in order for h^{max} to generate the representation of $S^R_{s_d}$, we have to recalculate the set of unreachable variables for all dead ends s_d after the unsolvability judgement.

In fact, almost always when both blind search and h^{max} were able to produce an inductive certificate, h^{max} is more memory intense. Only in few cases, h^{max} is as memory efficient as blind search, most likely due to no dead-ends in the search space.

Despite the larger memory consumption, in most cases, h^{max} fails far less due to memory exhaustion than blind search as can be seen in Table 9.3. This is due to pruning: in a planning task with many dead-end states, pruning permits h^{max} to keep a relatively low memory profile in comparison with blind search, as h^{max} has to keep far fewer states in memory at the same time.

9.3 Verification

Despite similar results in the generation of inductive certificates, we observe significant differences when it comes to verification.

Out of the generated certificates, $\mathrm{VER}^{\mathrm{Sp}}$ was able to verify 20% and 30%, $\mathrm{VER}^{\mathrm{BC}}$ verified 87% ad 92% for blind search and h^{max} and $\mathrm{VER}^{\mathrm{D}}$ managed to verify 71% for both search algorithms within limits. All verified certificates are valid.

	blind		h^{max}	
	memory	time	memory	time
$\mathrm{VER}^{\mathrm{Sp}}$	144	4	121	7
$\mathrm{VER}^{\mathrm{BC}}$	4	0	10	0
$\mathrm{VER}^{\mathrm{D}}$	50	0	45	0

Table 9.4: Reason for failures during verification in tasks where a certificate was generated

The verification of both heuristics mainly fails due to the memory exhaustion as can be seen in Table 9.4. One explanation for the high number of failures of $\mathrm{VER}^{\mathrm{Sp}}$ is on one hand the very high generation rate of $\mathrm{FD}^{\mathrm{Sp}}$ and on the other hand the reason that the entire "transformation" from certificate formula into many small subcalls takes place in the $\mathrm{VER}^{\mathrm{Sp}}$. Additionally, the verifier needs to store a large formula in memory in addition to the high number of calls to the SAT-solver. The high number of memory exhaustion for FD^{D} could be explained by the size of the validation formulas, as the certificates are on average quite large, whereas the low failures of $\mathrm{VER}^{\mathrm{BC}}$ are due to the high number of failures of $\mathrm{Trans}^{\mathrm{BC}}$.

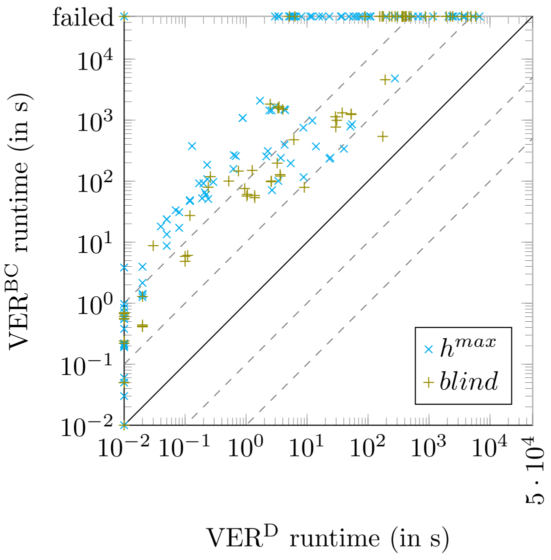
Although $\mathrm{VER}^{\mathrm{D}}$ has a comparatively high number of failures, it is still by far the most applicable, as it was capable of certifying around 48% of all tasks in which FD determined unsolvability, whereas $\mathrm{VER}^{\mathrm{Sp}}$ was able to validate about 23% and $\mathrm{VER}^{\mathrm{BC}}$ around 29%.

Figure 9.3a and Figure 9.3b show the verification time of $\mathrm{VER}^{\mathrm{BC}}$ and $\mathrm{VER}^{\mathrm{Sp}}$ in comparison with $\mathrm{VER}^{\mathrm{D}}$. Both $\mathrm{VER}^{\mathrm{BC}}$ and $\mathrm{VER}^{\mathrm{Sp}}$ are in all cases significantly slower than $\mathrm{VER}^{\mathrm{D}}$.

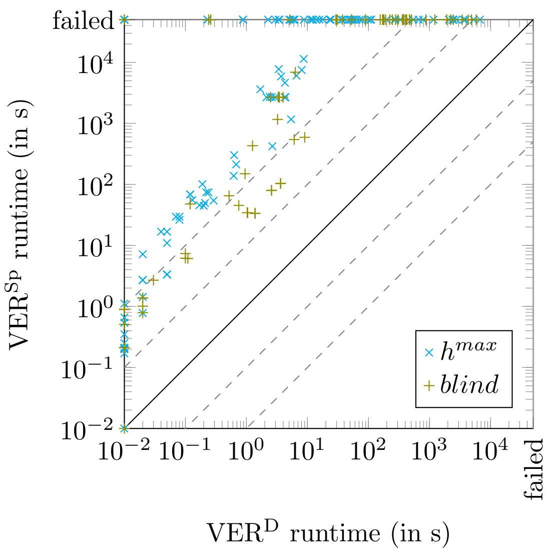
Comparing the verification time as a function of the size of the certificate size for both $\mathrm{VER}^{\mathrm{BC}}$ and $\mathrm{VER}^{\mathrm{D}}$ gives us an overview about the effectiveness of the verifiers. From Figure 9.4 we can deduce a positive correlation between validation formula size and required time to validate it and additionally we can see that generally $\mathrm{VER}^{\mathrm{D}}$ is far more effective. One reason for this is the use of the simple versus the generalized Tseitin transformation. Keep in mind that in the majority of cases the certificate of FD^{D} is smaller, as seen in Figure 9.1d, which makes this result even more impressive.

9.4 Summary

The experimental evaluation of the certifying approaches based on CNF certificates yields interesting results. Although both FD^{D} and $\mathrm{FD}^{\mathrm{BC}}$ are able to generate a formula in the majority of cases, the

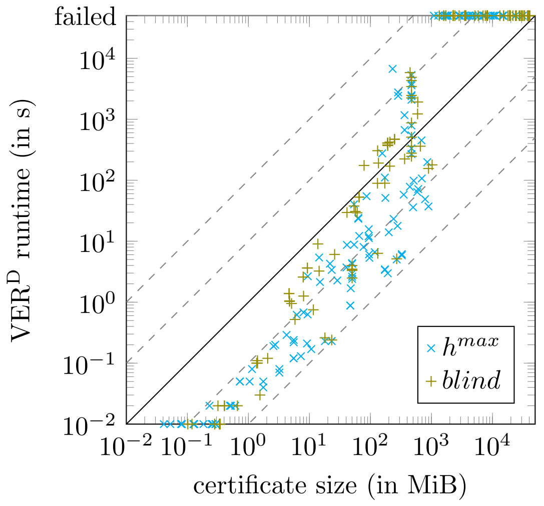


(a) Verification time of $\mathrm{VER}^{\mathrm{D}}$ vs $\mathrm{VER}^{\mathrm{BC}}$

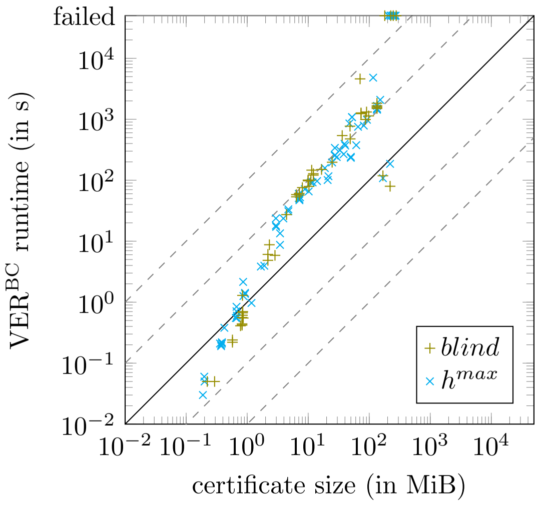


(b) Verification time of $\mathrm{VER}^{\mathrm{D}}$ vs $\mathrm{VER}^{\mathrm{Sp}}$

Figure 9.3: Verification time as a function of the certificate size



(a) FD^{D}



(b) $\mathrm{VER}^{\mathrm{BC}}$

Figure 9.4: Verification time as a function of the certificate size for $\mathrm{VER}^{\mathrm{D}}$ and $\mathrm{VER}^{\mathrm{BC}}$

success of $\mathrm{FD}^{\mathrm{BC}}$ rapidly decays. Because $\mathrm{FD}^{\mathrm{BC}}$ does not actually produce a validation formula in CNF but only a description of it, it needs to do a transformation step before being able to validate the formula. Unfortunately, this transformation is unsuccessful in most cases. In total, $\mathrm{FD}^{\mathrm{BC}}$ and $\mathrm{Trans}^{\mathrm{BC}}$ are able to generate a validation formula in about 30% of cases, whereas FD^{D} succeeded in 70%. In contrast to that, $\mathrm{FD}^{\mathrm{Sp}}$ is able to generate a certificate in 90% of cases. Comparing the certificate size yields that $\mathrm{FD}^{\mathrm{Sp}}$ generates the smallest certificates, whereas $\mathrm{Trans}^{\mathrm{BC}}$ produces in the majority of cases the largest. Although $\mathrm{VER}^{\mathrm{BC}}$ was able to verify around 90% of the generated certificates, the is result is diminished by the small number of generated certificates to begin with. $\mathrm{VER}^{\mathrm{Sp}}$ has only a very small verification rate of around 23% and is therefore for the most part not suitable. On the other hand, $\mathrm{VER}^{\mathrm{D}}$ validated around 70% of the presented certificates and was able to certify around 48% of all unsolvability judgments of FD. Hence, FD^{D} performed best under the considered approaches and is therefore the preferred choice of the three.

10 Comparison

In this chapter, we will compare the developed methods with the BBD based inductive certificates from Eriksson, Röger, and Helmert (2017). For this, we will use FD^{D} as a representative for the approach of expressing inductive certificates with CNF formulas. As we have seen in Chapter 9, FD^{D} has an overall higher number of successful generations and validations of the certificate and is generally the best performing out of the three approaches $\mathrm{FD}^{\mathrm{Sp}}$, $\mathrm{FD}^{\mathrm{BC}}$ and FD^{D}.
In contrast to FD^{D}, the inductive certificates developed in Eriksson (2019a) is represented by BDD's and is therefore denoted by $\mathrm{FD}^{\mathrm{BDD}}$.
Analogous to FD^{D}, $\mathrm{FD}^{\mathrm{BDD}}$ is also implemented as an extension of Fast Downward. This allows for a direct comparison between the two formalism CNF and BDD, whereas otherwise results might be diluted by employing different planning systems. Additionally, both FD^{D} and $\mathrm{FD}^{\mathrm{BDD}}$ make use of the same argument for unsolvability: inductive sets. Despite these similarities between the two, $\mathrm{FD}^{\mathrm{BDD}}$ does not produce a validation formula in CNF, hence we will use two separate verifiers: We can validate $\mathrm{FD}^{\mathrm{BDD}}$ with a validation tool developed for this specific purpose. As before, FD^{D} is validated with `Kissat`.
We compare the two approaches from a number of different perspectives. Following the list of important properties of certificate developed in Eriksson, Röger, and Helmert (2017), we focus on a general applicability, efficient generation and verification. Furthermore, we will investigate differences between the two approaches in their certificate size and the efficiency of the verifier with respect to the certificate size.

10.1 Generation

Both FD^{D} and $\mathrm{FD}^{\mathrm{BDD}}$ are relatively similar when it comes to a general applicability of certificates: FD^{D} is able to generate an inductive validation formula in about 72% of all cases where FD terminated with an unsolvability judgement, whereas $\mathrm{FD}^{\mathrm{BDD}}$ is successful in 60% for blind search and 74% for h^{max}, as can be seen in Table 10.1. Depicted in Table 10.2, we can see that FD^{D} fails almost always due to time for blind search, whereas $\mathrm{FD}^{\mathrm{BDD}}$ fails exclusively due to memory. This is in contrast to h^{max} where both certifying approaches fail in about one part to memory and about two parts due to time.
In almost all cases where both algorithms produce a certificate, $\mathrm{FD}^{\mathrm{BDD}}$ creates smaller files, as is presented in Figure 10.1. In some cases, in particular when comparing certificates for h^{max}, we observe a difference of an order of magnitude. However, the certificate size scales linear, meaning that

	FD	*blind* FD^{D}	*blind* VER^{D}	*blind* FD^{BDD}	*blind* VER^{BDD}	h^{max} FD	h^{max} FD^{D}	h^{max} VER^{D}	h^{max} FD^{BDD}	h^{max} VER^{BDD}
3unsat(30)	15	10	10	10	10	15	10	10	10	10
bag-barman(20)	12	8	0	4	4	8	8	0	4	4
bag-gripper(25)	3	2	2	2	2	2	2	2	2	2
bag-transport(29)	7	6	3	6	5	6	6	3	6	5
bottleneck(25)	10	8	6	8	8	21	16	14	17	15
cave-diving(25)	7	7	5	6	6	7	7	5	6	6
chessboard-pebbling(23)	5	4	3	4	4	5	4	3	4	4
diagnosis(13)	4	2	2	2	2	5	4	4	5	5
document-transfer(20)	5	5	3	4	4	7	6	5	6	6
mystery(9)	2	1	0	1	1	2	1	0	1	1
nomystery(150+24)	34	16	0	4	4	59	27	12	33	25
pegsol(24)	24	24	19	24	24	24	24	20	24	24
pegsol-row5(15)	5	4	4	4	3	5	4	4	4	4
rovers(150+20)	10	6	0	4	3	15	9	7	11	8
sliding-tiles(20)	10	10	0	10	10	10	10	10	10	10
tetris(20)	10	5	5	5	5	5	5	5	5	5
tpp(30+25)	24	15	9	14	12	23	14	9	15	12
total(697)	187	133	81	112	107	219	158	113	163	146

Table 10.1: Coverage comparison of FD^{D} and FD^{BDD} regarding generation and verification

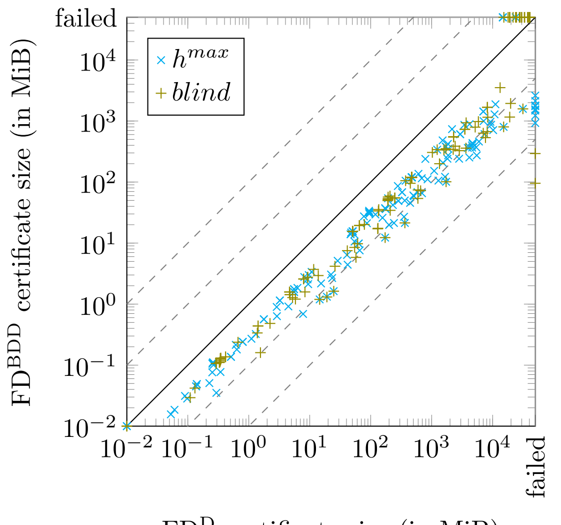
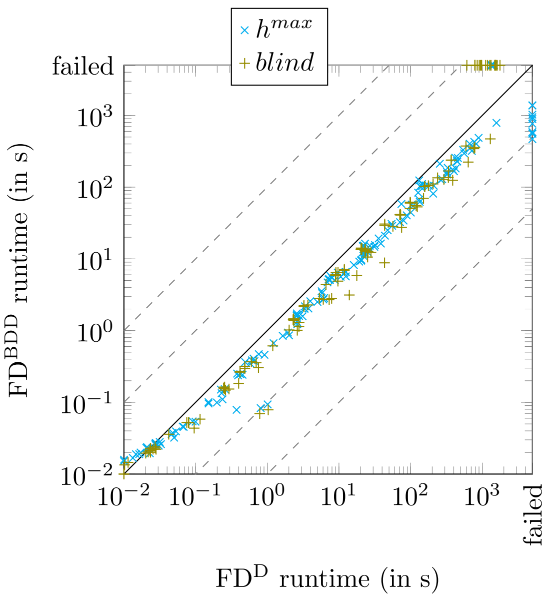


Figure 10.1: Comparison of certificate size of FD^{D} and FD^{BDD}

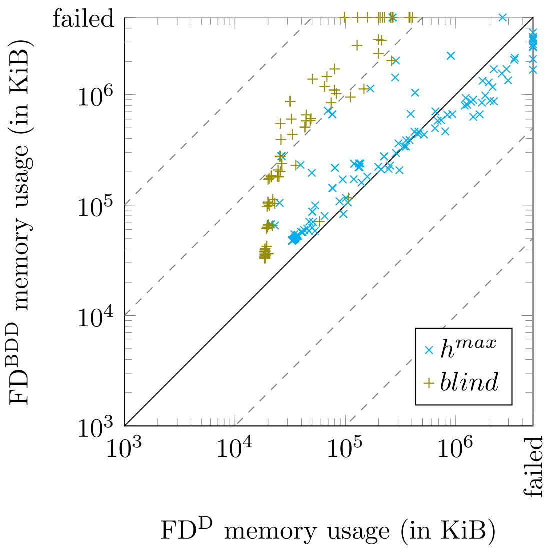
	blind memory	*blind* time	h^{max} memory	h^{max} time
FD^{D}	1	53	21	40
FD^{BDD}	54	0	21	35
VER^{D}	50	0	45	0
VER^{BDD}	0	5	0	17

Table 10.2: Reasons for failure in tasks that FD solved

if FD^{BDD} generates a large certificate, the certificate generated by FD^{D} is proportionally similar. One reason for this is, that BDD's support a very compact description of a formula, whereas FD^{D} does not simplify the validation formula itself. As BDD's represent propositional formula in most cases more compactly, the difference in certificate size is expected. Additionally, since file writing takes up a large portion of the generation process, it is no surprise that we see similar results in the time comparison depicted in Figure 10.2a: FD^{D} needs almost always more time than FD^{BDD}. Despite similar success



(a) time



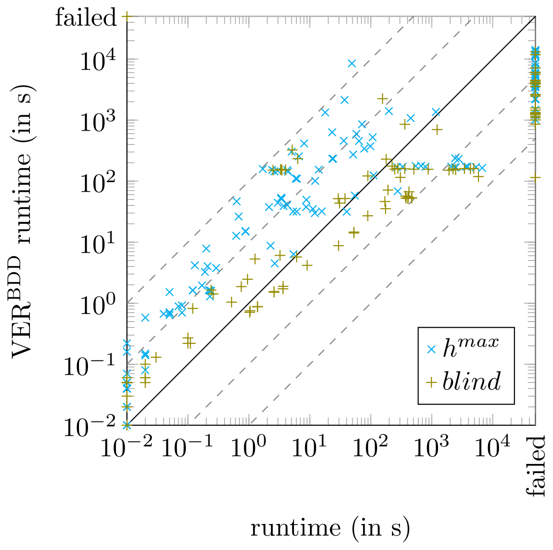
(b) memory

Figure 10.2: Comparison of FD$^{\text{D}}$ and FD$^{\text{BDD}}$

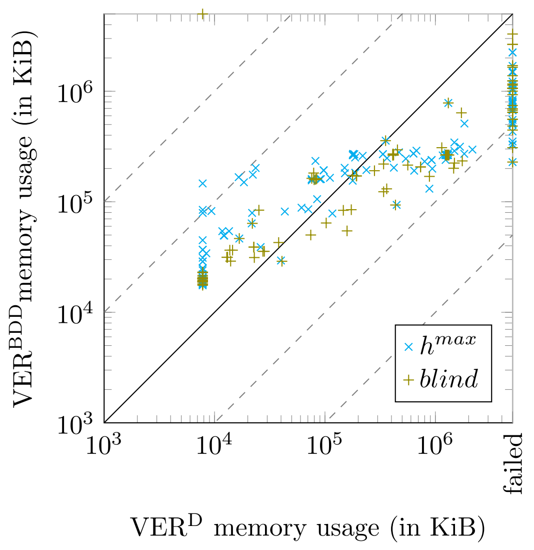
in the generation, both approaches fail on different problem instances, as can be seen in Figure 10.2a. In the case where FD$^{\text{BDD}}$ is able to generate a certificate, FD$^{\text{D}}$ fails exclusively for h^{max}, and FD$^{\text{BDD}}$ almost always fails for blind search. This could be due to the fact that adding a state to a BDD takes linear time, but is still not as performant as appending a state as CNF formula.
However, FD$^{\text{D}}$ and FD$^{\text{BDD}}$ are vastly different when it comes to memory consumption. Generally, we observe a large deviation between the two approaches. Whereas memory consumption for h^{max} is in most cases relatively balanced, FD$^{\text{D}}$ has a strong memory advantage for blind search. A comparison of FD$^{\text{BDD}}$ and FD$^{\text{D}}$ is presented in Figure 10.2b. The difference in memory consumption can be explained by the use of BDD's: whereas reduced BDD's permit a compact description, their generation and reduction is memory intense. In particular, FD$^{\text{BDD}}$ creates a BDD for each expanded state and for each dead-end. Since blind search does not determine any dead-ends, FD$^{\text{BDD}}$ needs to generate a BDD for each reachable state. Since FD$^{\text{D}}$ does not require additional computation for blind search, it is much more performant. In fact, as can be seen on the right side of Figure 10.2b, FD$^{\text{D}}$ is often able to generate a certificate in cases where FD$^{\text{BDD}}$ fails. Furthermore, FD$^{\text{BDD}}$ needs to store the entire BDD in memory and can only write it out after every state is represented through the BDD, whereas FD$^{\text{D}}$ is able to write the formula continuously. As both FD$^{\text{D}}$ and FD$^{\text{BDD}}$ rely on the same reachability analysis for dead-end formulas, they are in most cases similar for h^{max}.

10.2 Verification

While FD$^{\text{D}}$ and FD$^{\text{BDD}}$ are at least somewhat similar regarding the generation of certificates, the verification yields different results. VER$^{\text{D}}$ was able to successfully verify 67% of cases where FD$^{\text{D}}$ generated a formula, whereas VER$^{\text{BDD}}$ verified 90% of the generated certificates by FD$^{\text{BDD}}$ within time limit. All verified certificates by VER$^{\text{D}}$ and VER$^{\text{BDD}}$ are valid. The validation of FD$^{\text{D}}$ fails exclusively because of memory, while on the contrary VER$^{\text{BDD}}$ rarely fails at all. If it fails, however, it is always because of time.



(a) time



(b) memory

Figure 10.3: Comparison of VER$^{\mathrm{D}}$ and VER$^{\mathrm{BDD}}$

Generally, we observe a similar behavior when it comes to verification time: short and long verification times coincide. In Figure 10.3a we can see that VER$^{\mathrm{D}}$ is often faster than VER$^{\mathrm{BDD}}$: although they are relatively even when it comes to blind search, VER$^{\mathrm{D}}$ is faster in 85% of cases where both methods are able to verify their respective certificate. Particularly when both verifiers required a small amount of time, VER$^{\mathrm{D}}$ has the advantage. Furthermore, in cases where VER$^{\mathrm{D}}$ is slower, we often observe only a minor discrepancy, whereas VER$^{\mathrm{BDD}}$ is often multiple orders slower.

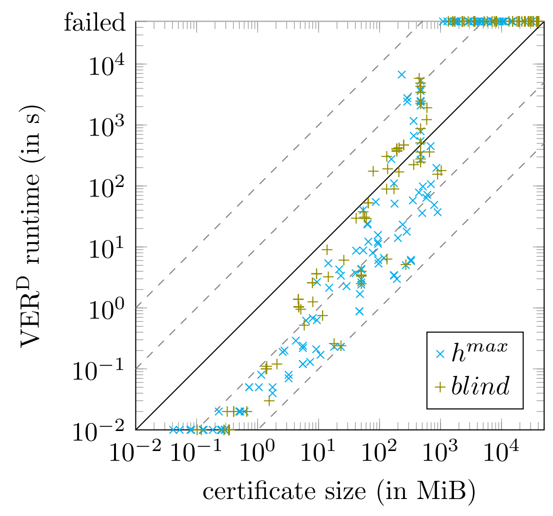
The memory consumption of VER$^{\mathrm{D}}$ and VER$^{\mathrm{BDD}}$ is a bit more mixed: similar as with time, VER$^{\mathrm{D}}$ is more performant for less intensive verification processes and has a lower memory consumption in about 60% of cases, where both approaches finished the validation process. However, as can be seen in Figure 10.3b, the memory usage of VER$^{\mathrm{D}}$ scales badly. Tasks where both approaches require a lot of memory are always more intense for the CNF based method. One explanation for this might be the **NP**-completeness when testing CNF formulas for satisfying assignments, whereas for BDD's we can generally test for satisfying assignments in polynomial time.

If we compare the runtime of the verifier as a function of the certificate size, we gain insight into the efficiency of the two different verifiers. An overview is given in Figure 10.4. Given the larger certificates and the overall faster verifying time, it is no surprise that VER$^{\mathrm{D}}$ is much more efficient. Generally, VER$^{\mathrm{D}}$ almost always able to stay on the lower part of the graph, especially for smaller certificates, which implies great efficiency of VER$^{\mathrm{D}}$ for smaller tasks. On the contrary, VER$^{\mathrm{BDD}}$ is in the majority of in the center of the chart, which corresponds to a comparatively low verification efficiency. However, VER$^{\mathrm{D}}$ does not scale as well as VER$^{\mathrm{BDD}}$ does, which can be deduced by the plots in Figure 10.4.

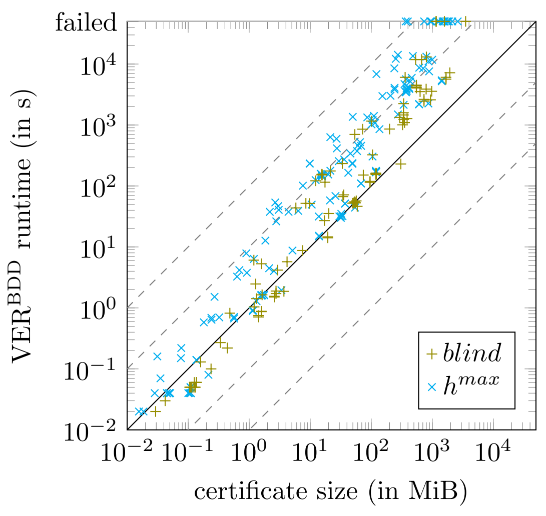
10.3 Summary

The experimental evaluation provides a reasonable argument for the use of CNF formulas as a representation of inductive certificates in comparison with the BDD based approach presented in Eriksson (2019a).

Generally, FD$^{\mathrm{D}}$ is able to keep up and sometimes even surpass FD$^{\mathrm{BDD}}$ both in generation and verification. FD$^{\mathrm{D}}$ outperforms FD$^{\mathrm{BDD}}$ minorly in general applicability and significantly in memory consumption for blind search in the generation.



(a) VER$^{\mathrm{D}}$



(b) VER$^{\mathrm{BDD}}$

Figure 10.4: Runtime comparison of VER$^{\mathrm{D}}$ and VER$^{\mathrm{BDD}}$ as a function of the certificate size

Regarding certificate size and runtime of the planner FD$^{\mathrm{D}}$ is in most cases only marginally worse than its contestant. When it comes to the validation of the certificate, VER$^{\mathrm{D}}$ is able to keep up with VER$^{\mathrm{BDD}}$ up to a certain point. Although VER$^{\mathrm{D}}$ is in most cases much faster during the verification and for small certificates more performant with regard to memory, VER$^{\mathrm{D}}$ generally does not scale as good as VER$^{\mathrm{BDD}}$. Certificates that require more memory or more time for both approaches, are verified with less memory by VER$^{\mathrm{BDD}}$.

Furthermore, VER$^{\mathrm{BDD}}$ is far more general: it is capable of validating almost all generated certificates, whereas VER$^{\mathrm{D}}$ is unable to verify about a third.

Therefore, FD$^{\mathrm{D}}$ and VER$^{\mathrm{D}}$ are able to outperform FD$^{\mathrm{BDD}}$ and VER$^{\mathrm{BDD}}$ only for certificates that require not too much memory to validate. Although CNF formulas are generally able to keep up with and even outperform the BDD formalism in some aspects, currently FD$^{\mathrm{BDD}}$ and VER$^{\mathrm{BDD}}$ remain the preferred choice.

11 Conclusion

In this thesis, we investigated the use of CNF formulas as a representation of inductive certificates for certified planning.

At first, we looked into how blind search is able to generate a formula representing all reachable states. By construction, this formula is in disjunctive normal form, and therefore we considered generating that formula from a regression perspective: representing the set of unreachable states allows us to obtain a CNF formula directly. With the help of this formula, we can introduce the notion of backwards inductive certificates. Similar to forward inductive certificate, their existence proofs unsolvability of the planning task. Although this seemed helpful at first, we derived that these two inductive certificates are actually equivalent.

Expressing the conditions of a forward or backwards inductive certificate as a single propositional formula allowed us to progress. The formula itself expresses the conditions of an inductive certificate in the sense that the planning task is unsolvable iff the formula is unsatisfiable. However, this is formula is not in CNF. To show unsatisfiability of the formula and therefore validate the certificate, we made use of (certifying) SAT-solvers that enable us to validate the formula in a certifying algorithm itself. Hence, we only have to translate the certificate into CNF and do not need to additionally develop a certifying verifier on our own.

As the inductive validation formula is not a CNF formula, we investigated two approaches: the first one exploits the fact that the formula is a disjunction of CNF formulas and hence it shows unsatisfiability by showing unsatisfiability of each disjunct. The second one transform the formula into CNF. However, as equivalence transformation are usually impractical and because we don't care about how but if the formula is satisfiable, we looked into satisfiability preserving transformations. For this, we utilized the Tseitin transformation, after which we finally derived a CNF formula as certificate.

We then extended this approach to delete relaxation heuristics, where instead of expanding all reachable states, we make use of pruning. However, not expanding dead-end states leads to lose inductivity of the certificate and thereby its completeness guarantee. By showing how to find and inductive set for each dead-end, we regained inductivity since all expanded states lead to expanded states and dead-ends and dead-ends are inductive.

The experimental evaluation compared three different approaches and shows a general applicability of the developed methods. Finally, we compared the best performing implementation with a BDD based certifying planning system. Although the CNF representation was able to keep up with the BDD approach regarding generation and outperform it in some aspects of the verification, it generally was only capable of matching the performance of the established approach up to a certain degree.

11.1 Future Work

One of the main drawbacks of CNF bases certificates in comparison with state-of-the-art certifying approaches is the comparatively low validation rate. Whereas the established certifying planning system based on BDD's can be verified in almost all cases where a certificate was generated, our approach was only able to validate every second certificate. Most often, the validation failed due to memory and hence it would be interesting to investigate more memory efficient certificates. For this, future work might look into simplifying the inductive certificate formula, as it is currently generated by explicitly appending each expanded state. On one hand, we could simplify the formula by only appending an equivalent but simpler representation by grouping states. For example, instead of appending the two formula $\varphi_1 = w \wedge \neg v$ and $\varphi_2 = w \wedge v$ individually, we could instead append $\varphi_{1,2} = w \wedge (\neg v \vee v) \equiv w$, which allows representing the two states much more compactly.
Similar, we could consider utilizing BDD's internally. As BDD's support important queries efficiently, we could make use of BDD's as an intermediate tool. Once we have derived a BDD expressing the inductive certificate, we could use it to generate a CNF formula from this reduced binary decision diagram. As the canonical representation of BDD's is based on reductions, these would correspond to simplification steps such as unit propagation in the resulting CNF formula. This would allow for potentially more compact certificate formulas in CNF. As the reduction is more memory intense than simply writing each state to the certificate, we would lose the memory efficiency of the current generation process, however we would potentially benefit from smaller certificate files and fewer generation time. Additionally, this could speed up the validation and allow us to efficiently certify more complex tasks.
Alternatively, we could consider the use of preprocessing tools for SAT-solvers. As the explicit representation introduces potential redundancies in the formula, a preprocessing tool would allow transforming a DIMACS CNF into an equisatisfiable but potentially simpler one.

Bibliography

Bäckström, C., Jonsson, P., and Ståhlberg, S. (2013). Fast Detection of Unsolvable Planning Instances Using Local Consistency. In: *Proceedings of the International Symposium on Combinatorial Search* 4.1, pp. 29–37.

Biere, A., Fazekas, K., et al. (2020). CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. University of Helsinki, pp. 51–53.

Biere, A., Heule, M., and Maaren, H. v. (2009). *Handbook of Satisfiability*. IOS Press.

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. In: *Artificial Intelligence* 129.1, pp. 5–33.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In: *Proceedings of the third annual ACM symposium on Theory of computing*. Association for Computing Machinery, pp. 151–158.

Darwiche, A. and Marquis, P. (2002). A Knowledge Compilation Map. In: *Journal of Artificial Intelligence Research* 17, pp. 229–264.

Eriksson, S. (2019a). Certifying Planning Systems: Witnesses for Unsolvability. PhD Thesis. University of Basel.

Eriksson, S. (2019b). Unsolvable PDDL Benchmarks.

Eriksson, S., Röger, G., and Helmert, M. (2017). Unsolvability certificates for classical planning. In: *Twenty-seventh International Conference on Automated Planning and Scheduling*, pp. 88–97.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. In: *Artificial Intelligence* 2.3-4, pp. 189–208.

Ganesh, V. and Vardi, M. Y. (2020). On the Unreasonable Effectiveness of SAT Solvers. In: *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, pp. 547–566.

Goldberg, E. and Novikov, Y. (2003). Verification of proofs of unsatisfiability for CNF formulas. In: *2003 Design, Automation and Test in Europe Conference and Exhibition*, pp. 886–891.

Helmert, M. (2006). The Fast Downward Planning System. In: *Journal of Artificial Intelligence Research* 26, pp. 191–246.

Hoffmann, J. and Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. In: *Journal of Artificial Intelligence Research* 14, pp. 253–302.

Junttila, T. A. and Niemelä, I. (2000). Towards an Efficient Tableau Method for Boolean Circuit Satisfiability Checking. In: *Computational Logic – CL 2000; First International Conference*. Springer, Berlin, pp. 553–567.

McConnell, R. M. et al. (2011). Certifying algorithms. In: *Computer Science Review* 5.2, pp. 119–161.

Seipp, J. et al. (2017). Downward Lab. URL: https://doi.org/10.5281/zenodo.790461.

Tseitin, G. S. (1983). On the Complexity of Derivation in Propositional Calculus. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Berlin, Heidelberg: Springer, pp. 466–483.

Vardi, M. Y. (2014). Boolean satisfiability: theory and engineering. In: *Communications of the ACM* 57.3, p. 5.

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis:	Certifying Unsolvability using CNF Formulas
Name Assessor:	Prof. Dr. Malte Helmert
Name Student:	Fabian Kruse
Matriculation No.:	2018-057-356

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: Rheinfelden, 07.02.2023 Student: F. Kruse

Will this work, or parts of it, be published?

No [✓ Yes.] With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: 07.02.2023

Place, Date: Rheinfelden, 07.02.2023 Student: F. Kruse

Place, Date: ______________________ Assessor: ____________________________

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis