hybr2d Theory of Hybrid Systems Informatik 2 | RWTH AACHEN UNIVERSITY

The present work was submitted to the LuFG Theory of Hybrid Systems

MASTER OF SCIENCE THESIS

EXPLAINING UNSOLVABLE PLANNING TASKS WITH UNSATISFIABLE CORES

Igor Nicolai Bongartz

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Gerhard Lakemeyer

Additional Advisor:
M.Sc. Francesco Leofante

Aachen,
December 23, 2018

Abstract

Explaining unsolvable planning tasks is a difficult challenge in the new field of *eXplainable AI*. We investigate an approach using the *Planning as Satisfiability* and *Bounded Model Checking* paradigm, together with minimal UNSAT cores of SMT. The generated minimal UNSAT cores are then used to explain unsolvable and solvable subtasks of the original planning task. Moreover, it is possible to state plan infeasibility by analyzing the minimal UNSAT cores. Finally, the presented approach is evaluated on a benchmark of well-known planning tasks.

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Acknowledgments

I am thankful for the opportunity to write this Master Thesis. Working on this academic work was a true challenge. The topic required creativity, formal specification of my ideas, their implementation and the empirical examination of the approaches. Facing these tasks required the complete knowledge, which I obtained during my Bachelor and Master studies over the last six years. Furthermore, I am confident that I will be able to apply those skills throughout the rest of my life.

I want to thank all the people which supported me during these nine months of research. First I mention Francesco Leofante. He was my adviser for this thesis and helped me with many obstacles. Next I thank all the people at the i2 who supported and assisted me, for example, while testing inside *SMT-RAT*. A big thanks goes to Erika Ábrahám, who made this work possible and to my second examiner Gerhard Lakemeyer.

Of course my family and friends have been a really helpful as well. Moreover, thank you Anna, Luke, Max, Pauline and Zoe, who despite their own work helped me out by reading through 82 pages of theoretical computer science gibberish.

And, as last time, at the end, thank you Saskia for your constant motivation, advice, warning, supply of sweets, Korean background music and everything everything else. But this time there is one more little human I need to mention. Dear Gustaf Noel, I want to thank you for being the boy you are.

Contents

Chapter 1

Introduction

Computer systems which imitate human actions are confusing if their actions do not fit to our own choices. In such cases a user might ask for an explanation for the performed or announced actions. Use-cases for such systems are *classification* (Machine Learning approaches which detect dogs on pictures[1]), *games* (AIs which play Go [SSS+17] or Chess [SHS+17]) or *plan execution* (planning the next steps of a semi-autonomous rover at a rescue site [KJK+12]). In order to trust the system, it is important to understand the decision making throughout its algorithm. The challenge is to return understandable and rational reasons instead of describing the algorithm itself [FLM17].

We will focus in particular on the field of *Planning*. A planning task contains a goal state which has to be reached by a sequence of actions from some initial state in a domain [ENS92]. The complexity of a planning task lies in the number and size of its fluents and operators. Solutions can be obvious to humans for simple domains but oftentimes they are difficult to detect due to the huge amount of possibilities. Planning in general has been studied for a long time and efficient planners have been developed for various planning task specifications at ICAPS[2] (planners generating plans as fast as possible or optimal plans).
Only recently the problem of unsolvable planning tasks has been raised in the planning community. The first *International Unsolvability Planning Competition* was organized at ICAPS 2016[3]. As a reaction, available planners have been modified or new approaches have been developed to detect unsolvable planning tasks (e.g., *Aidos* [SPS+16], *SymPA* [Tor16], *CLone* [SH16], and *iPlanProver* [SK16]).

Besides the development in the planning community, a new field of *eXplainable AI* (XAI) has been established[4]. The aim of XAI is to provide AI systems and models which can explain their actions. In terms of *Explainable Planning* (XAIP) this can contain the following information: 1. Why is an action necessary? 2. Why is a different action a bad choice? 3. Why is something not possible? As stated in [FLM17] an answer to the third question, the case of infeasibility for some planning task, is a

[1] https://www.kaggle.com/c/dog-breed-identification
[2] https://ipc2018-classical.bitbucket.io/#description
[3] https://unsolve-ipc.eng.unimelb.edu.au/
[4] http://icaps18.icaps-conference.org/xaip/

difficult challenge. This problem of XAI is the topic of this Master Thesis. To the best of our knowledge explaining unsolvable planning tasks has not been studied so far.

The approach we investigate uses tools from *SAT Modulo Theories* (SMT) to extract details about causes of infeasibility instead of only responding *»the planning task is unsolvable«*, and stating the infeasibility of planning tasks.
In SMT we evaluate satisfiability of logical formulas by searching for a satisfying assignment of variables, or proving that none exists. Following the *Planning as Satisfiability* paradigm [KSAH92], [KMS96] in this work we translate planning tasks into encodings of SMT problems. If the encoding is satisfiable it is possible to extract a plan solving the corresponding planning task. In contrast, unsatisfiability of the encoding states only that no plan of a specific length solves the given planning task, but plans of different length might exist. It is possible to extract *minimal unsatisfiable* (UNSAT) *cores*[5] of the original unsatisfiable encoding to identify parts of the planning task which are solvable and unsolvable.
Moreover, it is possible to state infeasibility of the given planning task. Referring to the *Bounded Model Checking* (BMC) paradigm [Bie09], we enroll the transition scheme of the planning task from 1 up to a predetermined bound, and only if no plan solving the planning task can be found we conclude infeasibility. Even if most efforts have been exerted on efficiently finding solutions for solvable planning tasks [Rin12], recently interest increased in improving to show infeasibility of planning tasks with BMC as well. This is achieved, for example, by over-approximating the set of reachable states in each enrolling of the transition scheme and once only dead ends are found infeasibility is stated [Sud14], [SK16].
We propose a technique called ***B**MC using **UNS**AT cores* (BUS) to state infeasibility of planning tasks making use of the generated minimal UNSAT cores of an unsatisfiable encoding [CGS11] for each enrolling. The generated cores are then checked for witnesses of infeasibility of planning tasks, describing a conflict in the transition scheme, which prohibits reaching the goal from the initial configuration [Str04]. The gathered information states problems rising from leaving the initial and reaching the goal configuration, and is similar to the reachability information in [Sud14], [SK16] but achieved in a different way as there is no need to maintain additional information. Due to the evaluation in a forward and backward fashion, the worst-case number of BMC iterations is reduced by a factor of 2, as in other tools using bidirectional search [Tor16]. However, generating all minimal UNSAT cores is exponential in the size of the encoding and increases the solving time of the original BMC algorithm.

The presented thesis is structured as follows: First the necessary preliminaries about SMT, Planning and XAI are presented in Chapter 2. Next, Chapter 3 contains the definition and properties of BUS and Chapter 4 describes how to use UNSAT cores for XAI. Evaluations of three experiments can be found in Chapter 5. We test minimal UNSAT core generation algorithms in Section 5.1, decide planning tasks with the proposed BUS approach in Section 5.2, and explain planning tasks with the tool `PEX` (*Planning EXplainer*) encapsulating BUS in Section 5.3. Finally, we discuss and conclude the thesis in Chapter 6.

[5] Later in Section 2.1.4 we will refer to minimal UNSAT cores as *minimal unsatisfiable subsets* (MUSs) and use this description throughout the thesis.

Chapter 2

Preliminaries

The work in the presented Master Thesis is based on three fields of research: *SAT Modulo Theories* (SMT), *Planning* and *eXplainable AI* (XAI). We investigate the challenge, which arises if we try to explain unsolvable planning tasks to users. As stated in [FLM17] it is already a difficult challenge to prove infeasibility of planning tasks because an exhaustive search of the state space has to be performed [Tor16]. In addition to investigate infeasibility of planning tasks we focus on explaining the causes of infeasibility in order to provide useful information, which enables the user to perform necessary actions. The tools are from the field of SMT (Section 2.1) while the general problem we are tackling is based on Planning (Section 2.2) and XAI (Section 2.3).

2.1 SAT Modulo Theories

The field of SMT deals with *first order logic* (FOL) formulae and is based on the *Satisfiability Problem* (SAT) which we explain first. Then we present the necessary extensions of SAT for the SMT blueprint. If the SAT/SMT procedure states unsatisfiability of a formula it is possible to extract a part of the formula which is already unsatisfiable, a so-called *unsatisfiable* (UNSAT) *core*. Several sophisticated algorithms exist to generate UNSAT cores. For this thesis we take a closer look at the algorithms *CAMUS* [LS08] and *Basic Linear Search* (BLS) [MSHJ$^+$13], which generate all *minimal* UNSAT cores.

2.1.1 First Order Logic (FOL)

FOL is based on a set of an infinite number of symbols grouped into logical and non-logical symbols [End01]. Logical symbols are parentheses (,), logical connective symbols $\wedge$,$\neg$, and variables x,y,$\ldots$. Non-logical symbols are quantifiers $\forall$, n-ary *predicates*, and n-ary *functions* with $n \in \mathbb{N}_0$. A specification of the non-logical symbols is called a *language*. We define finite sequences of symbols (so-called *expressions*):

- A 0-ary function is called a *constant*.
- A *term* is obtained by applying the n-ary *term-building operation* $\mathcal{F}_f(t_1,\ldots,t_n) = f\ t_1,\ldots,t_n$ zero or more times with t_i being a constant, a variable or a term itself.

- An n-ary predicate $P\ t_1, \ldots, t_n$ with t_i a term is called an *atom*.
- *Formulae* are atoms (or formulae) combined via quantifiers and logical connective symbols. A formula which contains only variables bonded by quantifiers is a *sentence*. Variables which are not bonded are called *free variables*.
- An atom or its negation are called *literals*.
- A disjunction of literals is called a *clause*.

Any so far defined expression is called *grounded* if no free variables occur in it. A conjunction of clauses defines a formula in *conjunctive normal form* (CNF). Equisatisfiable CNF formulae can be provided for every formula in polynomial time via Tseitin's encoding [Tse83]. Clauses can be seen as *constraints* and a formula as a *constraint system* [LS08].

A *structure* $\mathfrak{A}$ provides a non-empty domain $|\mathfrak{A}|$ and uses it to define constants, predicates and functions. Constants are elements of $|\mathfrak{A}|$. Every n-ary predicate $P^{\mathfrak{A}} \subseteq |\mathfrak{A}|^n$ is a subset of the domain. Every n-ary function $f^{\mathcal{A}} : |\mathfrak{A}|^n \to |\mathfrak{A}|$ maps into the domain.
An *assignment* $a : V \to |\mathfrak{A}|$ maps each free variable $v \in V$ of a formula into the domain. Once all free variables are replaced by their specification $a(v)$ we can check whether the obtained sentence is *satisfied* in $\mathfrak{A}$ (we refer to [End01] for a formal definition of satisfaction). In case no assignment yields a sentence, which is satisfied in $\mathfrak{A}$ the formula is *unsatisfied*. If a formula with free variables is satisfied in $\mathfrak{A}$ for all assignments, we call this a *tautology*. A structure $\mathfrak{A}$ is a *model* (indicated by $\models_{\mathfrak{A}} \varphi$) of a sentence φ if for all assignments φ is satisfied with respect to $\mathfrak{A}$.
We introduce the concept of *logical implication* $\Gamma \models \varphi$ with a set of formulae Γ and a formula φ as a statement that every structure which is a model of every formula $\gamma \in \Gamma$ together with an assignment a is a model of φ with a as well. A *theory* T is a set of sentences closed under logical implication (for every $t \in T$ the statement $T \models t \Rightarrow t \in T$ holds).
FOL is *undecidable* (in fact *semi-decidable*) for some more expressive theories (e.g., *Peano arithmetic*). *Decidable* fragments of FOL exist (e.g., *Pressburger arithmetic* and *propositional logic* (PL) [SW97]), balancing expressibility and tractability. PL contains only Boolean variables and connective symbols, and omits quantifiers, predicates, and functions.

2.1.2 Satisfiability Problem (SAT)

SAT is used in many applications, thanks to its improving efficiency [CGS11]. The procedure is used to determine satisfiability of PL formulae [End01]. If an input formula is *satisfiable* (SAT), an assignment for each variable can be found such that the overall formula resolves to *TRUE*. Otherwise it can be stated, no assignment satisfies the formula, making it *unsatisfiable* (UNSAT).
Although the problem is NP-complete, as stated by the *Cook Levin theorem* [Coo71], well performing algorithms exist. The *Davis-Putnam-Logemann-Lovecup* (DPLL) algorithm is at the core of state-of-the-art SAT solvers [DLL62]. It uses the techniques of *enumeration*, *boolean constraint propagation* (BCP), and *conflict resolving*. A definition of the algorithm can be found in Algorithm 1.

Algorithm 1 DPLL

```
1: procedure DPLL(φ)
2:     while true do
3:         while ¬BCP() do
4:             if ¬resolveConflict() then
5:                 return UNSAT
6:         if ¬decide() then
7:             return SAT
```

First, we apply method `BCP` on the formula to detect, if the initially empty partial assignment enforces other variables to be assigned to a certain value. Once `BCP` terminates, we check if a conflict occurred and, if yes, whether it can be resolved. Resolving with `resolveConflict` uses techniques of resolution, clause learning and backtracking to detect which assignments on which level are responsible for a conflict. If a conflict cannot be resolved we know that the formula is UNSAT. Otherwise we continue with `decide`. This method searches, based on heuristics, for the next unassigned variable. If no variable can be found it means we found a satisfying assignment and return SAT [CGS11].
The SAT problem is often used to show for other constraint satisfaction problems via polynomial reduction their membership to the NP-complete problems [LS08].

Example 2.1.1 (PL formulas)**.** *Consider the following PL formulas:*

$$\begin{aligned} \varphi &= (x) \wedge (\neg x \vee y) \\ \varphi' &= (x) \wedge (\neg x \vee y) \wedge (\neg y) \end{aligned}$$

The first formula φ is satisfiable with the assignment $\{x \to TRUE, y \to TRUE\}$. The second formula φ' is unsatisfiable as the first and third clauses require x to be TRUE and y to be FALSE, but then the second clause is not satisfied.

2.1.3 SAT Modulo Theories (SMT)

Similar to SAT we are interested in satisfiability, but this time for FOL formulae [End01]. Therefore the SMT procedure is a combination of SAT and some decision procedures for specified theories (standard descriptions of such theories are provided by *SMT-LIB*[6]). Throughout this thesis, besides the class of PL formulas, we are interested in the SMT logic of *QF_LIRA*, *QF_LRA* and *QF_LIA*. These classes consist of *quantifier-free linear real* and/or *integer arithmetic* formulae and serve as specification languages to translate planning task into SMT problems.

Here we describe the *lazy* SMT procedure for an input formula φ [BSST09]. We alternate between the decision procedure for the *propositional skeleton* $\varphi_{skeleton}$ of the input formula and for the considered theory. The skeleton is obtained by replacing each clause with a fresh propositional variable. If the SAT procedure states that $\varphi_{skeleton}$ is already unsatisfiable, we return UNSAT. In case that a satisfying assignment is found, the *activated* clauses must be consistent in the theory. Therefore, a

[6] http://smtlib.cs.uiowa.edu/index.shtml

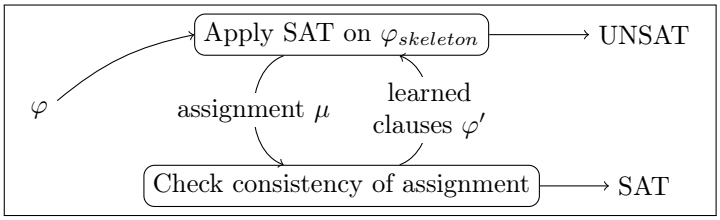


Figure 2.1: Visualized SMT procedure

decision procedure for the corresponding theory has to be applied (e.g., *Simplex* for *QF_LRA* [DNS05]). If the clauses set to *TRUE* and the negation of the clauses set to *FALSE* do not contradict each other we return SAT. Otherwise we add a clause describing the conflict found on the theory level and preventing SAT from finding the same assignment again. This alternation is executed up to the point that a satisfying assignment has been found or all assignments are proven to be inconsistent. A visualization of SMT can be found in Figure 2.1.
Examples of SMT solvers are *MiniSAT* [ES03], *SMT-RAT* [CKJ+15] and *Z3* [DB08]. In order to compare SMT solvers benchmarks have been created based on the SMT-LIB format. For example, a collection of benchmarks containing several theories can be found on the web-based service *StarExec* [7].

Example 2.1.2 (FOL formulas)**.** *Consider the following FOL formulas from the theory QF_LRA:*

$$\begin{aligned} \varphi &= (0 \leq x) \wedge (x \leq 10) \\ \varphi' &= (x \leq 0) \wedge (10 \leq x) \end{aligned}$$

The first formula φ is satisfiable with the assignment $\{x \to 5\}$. The second formula φ' is unsatisfiable because no real number fulfills both clauses.

2.1.4 UNSAT Cores

From this point on, unless stated otherwise, we use φ as the set of clauses equivalent to a FOL formula. If a given set φ of clauses is stated unsatisfiable it is possible that a subset $\varphi' \subseteq \varphi$ of clauses is already unsatisfiable as well. This subset of clauses is called *UNSAT core* and can be *minimal* or a *minimum*. A UNSAT core is minimal if removing any clause yields a satisfying subset of clauses. It is possible that several minimal UNSAT cores exist. A smallest of those with respect to the number of clauses is called a minimum.

Different techniques exist on how to generate UNSAT cores. Apart from using φ as a trivial UNSAT core, more sophisticated approaches return smaller UNSAT cores. For example, the resolution graph can indicate a smaller subset of clauses belonging to a UNSAT core, as used in the Z3 solver [DB08]. However, this method does not necessarily return a minimal UNSAT core.
In order to generate a minimal UNSAT core, more dedicated techniques have to be

[7] https://www.starexec.org/starexec/secure/explore/spaces.jsp?id=294832

Algorithm 2 Propagate Choice of Set and Element

```
1: procedure PROPAGATECHOICE(C,C,C_i)
2:     for all C' ∈ C do
3:         if C_i ∈ C' then
4:             C = C \ {C'}
5:     for all C'_i ∈ C do
6:         for all C' ∈ C do
7:             if C'_i ∈ C' then
8:                 C' = C' \ {C'_i}
9:     C ← removeSupersets(C)
10:    return C
```

used. General approaches are *addition* or *deletion*. These techniques start with an empty satisfiable or the complete unsatisfiable set of clauses and iteratively add or remove clauses up to a point the subset is un- or satisfiable. Specialized approaches for more complex algorithms as *mixed-integer* or *nonlinear programming* exist [Chi08].
The next step is not only to generate a single minimal UNSAT core, but to enumerate all of them. The general principle of enumerating can be applied here while using one of the above techniques as a procedure to generate the next minimal UNSAT core. For each detected minimal UNSAT core we add a blocking clause, prohibiting to generate the same minimal UNSAT core again. If the procedure states that no more minimal UNSAT core can be found, we know that we detected all of them. However, brute force enumerating is inefficient, because the worst case yields an exponential number of minimal UNSAT cores in the size of clauses.
We can make use of the duality between all *minimal correction subsets* (MCSs) and *minimal unsatisfiable subsets* (MUSs aka minimal UNSAT cores) via their hitting set definition. An MCS defines a set of clauses, which once removed from φ yield a satisfiable subset of clauses. Each MUS is an irreducible hitting set of all MCSs and each MCS is one of all MUSs [LS08]. For a hitting set (HIT) H of a collection of sets $\mathcal{C}$ it holds that for each set $C \in \mathcal{C}$ at least one element $C_i \in C$ is in H. A HIT is irreducible if no element can be removed without violating the hitting set property from H. The complement of an MCS is a *maximum satisfying subset* (MSS)[8]. Techniques generating all MCSs are more efficient than generating all MUSs and speed up the procedure. Two approaches based on this observation are *CAMUS* [LS08] and *Basic Linear Search* (BLS) [MSHJ+13]. Because both aim to generate all MUSs via the hitting sets of all MCSs, they only differ in their MCSs generation. All algorithms will be presented in detail.

Hitting Set Generation

An irreducible hitting set (iHIT) H from a set of sets $\mathcal{C}$ is computed by iteratively choosing a set $C \in \mathcal{C}$ and an element from this set $C_i \in C$ [LS08]. In our case C_i is a clause, C is a set of clauses and $\mathcal{C}$ is a set of set of clauses.
The element C_i is then added to H and two modifications on $\mathcal{C}$ are performed. We say that we *propagate the choice* of C_i and C in $\mathcal{C}$ (Algorithm 2). First from Line 2 on, all sets $C' \in \mathcal{C}$ with $C_i \in C'$ are removed from $\mathcal{C}$. Next from Line 5 on, all remaining

[8]The abbreviations MCS, MSS and MUS were introduced in [LS08]. From this point on we will use this notation for consistency.

Algorithm 3 Irreducible Hitting Set Generation

```
1: procedure HITTINGSETGENERATION(C,H)
2:     if C = ∅ then
3:         print(H)
4:     else
5:         for all C ∈ C do
6:             for all C_i ∈ C do
7:                 H' = H ∪ C_i
8:                 C' ← propagateChoice(C,C,C_i)
9:                 hittingSetGeneration(C',H')
```

elements from the set $C \setminus C_i$ are removed from all other remaining sets $\mathcal{C}$. After these modifications it has to be ensured that no sets are supersets of other sets.

In order to generate all iHITs $\mathcal{H}$ we have to apply the above principle to a recursive algorithm, where we loop independent of the order over all possible sets and elements. The break condition of the algorithm is an empty $\mathcal{C}$ and then we print the detected iHIT H(Algorithm 3). The input is the set of sets $\mathcal{C}$ and a currently empty iHIT H. Once the initial call of the algorithm terminates all iHITs of $\mathcal{H}$ have been printed.

Example 2.1.3 (Hitting Set Generation)**.** *Assume we have the following set of sets:* $\mathcal{C} := \{\{C_1\},\{C_2,C_3\},\{C_2,C_4\}\}$. *All iHITs are:* $\mathcal{H} = \{\{C_1,C_2\},\{C_1,C_3,C_4\}\}$.

This hitting set generation algorithm can only be applied on $\mathcal{C}$ if no set $C_i \nsubseteq C_j$ is a subset of another with $C_i,C_j \in \mathcal{C}$. Otherwise the second modification can remove sets which are subsets of a chosen set and H would be no iHIT anymore. For example, let us consider $\mathcal{C} := \{\{C_1,C_2\},\{C_2\}\}$ with the correct iHITs $\mathcal{H} = \{C_2\}$. However, choosing C_1 of the first set of clauses first, would cause the second set of clauses $\{C_2\}$ to disappear, thus resulting into a false iHITs $\mathcal{H} = \{\{C_1\}\}$

CAMUS

The algorithm CAMUS [LS08] (representing the phrase *Compute All Minimal Unsatisfiable Subsets*) can be found in Algorithm 4.
First we apply the method `getSelector`, such that all clauses C_i are replaced by C_i' defined as $\neg s_i \vee C_i$ with clause-selector variables s_i. This can be seen as an implication $s_i \rightarrow C_i$ allowing to activate or deactivate the underlying clause C_i. By simply deactivating all clauses we obtain a satisfiable set of clauses.
Next, we increase a variable k, indicating the number of clauses which are allowed to be deactivated. This is realized with an *atmost* constraint, enforcing that only at most k many clause-selector variables are *FALSE*. This constraint can be handwritten, encoded by a cardinality encoder or handled internally via watching literals and returning UNSAT, if more than k selector variables are set to *FALSE*. All deactivated clauses indicate an MCS, because removing those clauses yields a satisfiable set of clauses.
We remember already found MCSs by asserting *blocking clauses* (BCs). For some MCS $M := \{C_1,\ldots,C_n\}$ we add the blocking clause $\varphi_{BC_{CAMUS}} := \bigvee_{C \in M} s_C$ to the set of blocking clauses φ_{BCs}. This clause assures that at least one of the clauses

Algorithm 4 CAMUS

```
1: procedure CAMUS(φ)
2:     φ_selector ← getSelector(φ)
3:     φ_BCs = ∅
4:     k = 0
5:     while solve(φ_selector ∪ φ_BCs) do
6:         k++
7:         φ_atmost_k = φ_selector ∪ atmost(k)
8:         while solve(φ_atmost_k ∪ φ_BCs) do
9:             Model m ← getModel(φ_atmost_k ∪ φ_BCs)
10:            φ_BCs = φ_BCs ∪ getBlockingClause(M)
11:    return hittingSetGeneration(getMCSs(φ_BCs))
```

belonging to an MCS must be activated, prohibiting that this MCS appears again. Once the set of clauses, extended by the atmost constraint, is unsatisfiable we know that no MCS of size k exists anymore. If $\varphi_{selector}$ extended by the already found blocking clauses φ_{BCs} is still satisfiable more MCSs of size $k' > k$ must exist and we increase k by one. Else we extracted all MCSs and can make use of the hitting set duality to generate all MUSs.

Example 2.1.4 (CAMUS on a PL formula)**.** *For this example, we extend the formula φ' from Example 2.1.1 to*

$$\varphi'' = \overbrace{(x)}^{C_1} \wedge \overbrace{(\neg x)}^{C_2} \wedge \overbrace{(\neg x \vee y)}^{C_3} \wedge \overbrace{(\neg y)}^{C_4}$$

with the indices representing the clauses. This updated formula is unsatisfiable as well because the original formula φ' is already unsatisfiable. In the first step we extend each clause by a selector variable and obtain a set of clauses

$$\varphi_{selector} = \texttt{getClauses}((\neg s_1 \vee x) \wedge (\neg s_2 \vee \neg x) \wedge (\neg s_3 \vee \neg x \vee y) \wedge (\neg s_4 \vee \neg y))$$

with `getClauses(φ)` *extracting the set of clauses from a FOL formula φ. The first call of* `solve(`$\varphi_{selector} \cup \varphi_{BCs}$`)` *is true as we can simply deactivate all clauses (by setting s_1,s_2,s_3,$s_4 \rightarrow FALSE$).*
Now we add the `atmost(1)` *constraint and check if only deactivating one clause yields a satisfying set of clauses. This is true indeed because we can deactivate C_1 (representing the MCS $\{C_1\}$) and obtain a satisfying assignment x,$y \rightarrow FALSE$. The blocking clause s_1 is then added to the complete set of blocking clauses and ensures that C_1 is never deactivated in the future again.*
Next, we increase k up to 2, because no other MCS of size 1 can be found. In this iteration of the outer loop we find the MCSs represented by the blocking clauses $s_2 \vee s_3$ and $s_2 \vee s_4$. No other MCSs exist and we collected all MCSs: $\{\{C_1\}, \{C_2,C_3\}, \{C_2,C_4\}\}$. After generating all hitting sets we obtain all MUSs as shown in Example 2.1.3: $\{\{C_1,C_2\},\{C_1,C_3,C_4\}\}$.

Algorithm 5 Basic Linear Search (BLS)

```
1: procedure BLS(φ)
2:     φ_BCs = ∅
3:     while solve(φ_BCs) do
4:         φ_MCS = ∅
5:         φ_MSS = φ_BCs
6:         for all clause ψ ∈ φ do
7:             if solve(φ_MSS ∪ ψ) then
8:                 φ_MSS = φ_MSS ∪ ψ
9:             else
10:                φ_MCS = φ_MCS ∪ ψ
11:        φ_BCs = φ_BCs ∪ getBlockingClause(φ_MCS)
12:    return hittingSetGeneration(getMCSs(φ_BCs))
```

Basic Linear Search (BLS)

The algorithm BLS [MSHJ+13] can be found in Algorithm 5. The aim is to construct all MSSs and extract the corresponding MCSs. First the current MSS is initialized with all up to this point detected blocking clauses. The current MCS is set to the empty clause. Then we iterate over all clauses in φ and check whether the current clause is satisfiable together with the current MSS. If the solving procedure states SAT we extend the current MSS by that clause. Else it is added to the current MCS. After all clauses in φ have been considered, we obtain a new MCS with respect to the blocking clauses. We apply this procedure as long as the set of blocking clauses is satisfiable.
For each found MCS we assert the corresponding blocking clause. For a given MCS $M := \{C_1, \ldots, C_n\}$ we add the blocking clause $\varphi_{BC_{BLS}} := \bigvee_{C \in M} C$. Note the difference compared to CAMUS, where we use the clause-selector variables for the blocking clause.
A possible improvement is called *caching* [PM17]. For each UNSAT solver call we store the returned UNSAT core. If a superset of any UNSAT core is about to be tested we avoid it. This is possible due to Proposition 1 in [PM17].

Example 2.1.5 (BLS on a PL formula)**.** *For this example, we use the same formula φ'' as in Example 2.1.5.*

$$\varphi'' = \overbrace{(x)}^{C_1} \wedge \overbrace{(\neg x)}^{C_2} \wedge \overbrace{(\neg x \vee y)}^{C_3} \wedge \overbrace{(\neg y)}^{C_4}$$

Assume we chose the clauses `getClauses` *(φ'') in order of their indices. Then, in the first iteration we obtain $\varphi_{MSS_1} = C_1 \cup C_3$ and $\varphi_{MCS_1} = C_2 \cup C_4$. The set of clauses φ_{MCS_1} corresponds to the MCS $\{C_2, C_4\}$ and yields the blocking clause $C_2 \vee C_4$.*
The next iteration yields $\varphi_{MSS_2} = \varphi_{BCs} \cup C_1 \cup C_4$ and $\varphi_{MCS_2} = C_2 \cup C_3$ corresponding to the MCS $\{C_2, C_3\}$ and the blocking clause $C_2 \vee C_3$.
Then we obtain $\varphi_{MSS_3} = \varphi_{BCs} \cup C_2 \cup C_3 \cup C_4$ and $\varphi_{MCS_3} = C_1$ corresponding to the MCS $\{C_1\}$ and the blocking clause C_1.
No other MCSs exist because φ_{BCs} is now unsatisfiable and we collected all MCSs:

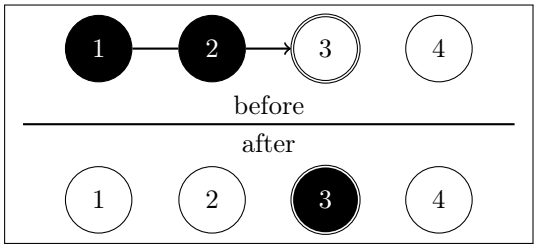


Figure 2.2: Visualized solvable pegsol-invasion instance. Locations which need to be filled are marked with a second circle. Initially filled locations are black.

$\{\{C_1\},\{C_2,C_3\},\{C_2,C_4\}\}$. *After generating all hitting sets we obtain all MUSs as shown in Example 2.1.3:* $\{\{C_1,C_2\},\{C_1,C_3,C_4\}\}$.

2.2 Planning

The planning problem has challenging variations. We could be interested in an *optimal* plan (with respect to some metric) or just use the first sound plan, which we are able to detect. Besides detecting plans for solvable planning tasks, the problem of proving infeasibility of planning tasks, which cannot be solved by any plan, exists. In Section 2.2.1 we first present the definition of planning tasks. Next in Section 2.2.2 we argue about decidability of planning tasks in order to understand which subset of planning tasks we can tackle later on. Finally, we list state-of-the-art planners for the problem of detecting plans for solvable planning tasks and stating infeasibility of planning tasks in Section 2.2.3.

2.2.1 Definitions

Before moving to more practical definitions of a planning task we take a closer look from a theoretical view upon it. Then we present the PDDL definition, a standard description language, encoding a planning task in two files, describing its domain and problem [KBC+98]. The third definition follows the *Planning as Satisfiability* paradigm [KSAH92], [KMS96] encoding a planning task in a FOL formula, arguing about its satisfiability and obtaining a plan as a side product.

Example 2.2.1 (Pegsol-Invasion)**.** *Pegsol-invasion is one of the domains which we later use in the evaluation in Chapter 5. The task is to fill a set of locations by making use of the following rule: If three locations X,Y,Z are a line, X,Y are filled and Z is free, then Z can be filled by clearing X and Y (we jump from X over Y to Z). A visualization of a solvable instance can be found in Figure 2.2 with four locations* l_1,l_2,l_3,l_4. *Initially* l_1,l_2 *are filled,* l_3,l_4 *are free and* l_3 *is desired to be filled.*
This instance can be turned unsolvable, if we desire to fill l_4 *instead of* l_3.

Formal

We use the formal definition for planning tasks from [ENS92]. Let $\mathcal{L}$ be a FOL language with finitely many constants, predicates and functions and a supply of an infinite number of variables. A *state* is a set of grounded atoms from $\mathcal{L}$. A grounded

atom a is true in state S if $a \in S$. Otherwise, a is false in S. An *action* is defined by a name $\alpha(X_1, \ldots, X_n)$ (α a description, and $X_1, \ldots, X_n$ denoting n input variables) and three lists. Among those lists is one list of literals (preconditions Pre_α) and two lists of atoms (additions Add_α, and deletions Del_α) following the STRIPS-blueprint [FN71]. Preconditions represent what needs to be fulfilled in order to execute the action, and additions and deletions state how executing the action affects the environment. Atoms which are modified by actions can be described as *fluents* of the domain.
An action $\alpha(X_1, \ldots, X_n)$ is *θ-executable* in a state S if a substitution θ mapping all variables to grounded terms exists, which fulfills two conditions:

- $\{\theta(A) | A \in Pre_\alpha\} \subseteq S$ (all positive atoms need to be in S)
- $\{\theta(B) | \neg B \in Pre_\alpha\} \cap S = \emptyset$ (no negated atoms are allowed in S)

Then the state after applying α is defined as $S' := (S - \theta(Del_\alpha)) \cup \theta(Add_\alpha)$ and we write $S \overset{\alpha,\theta}{\Longrightarrow} S'$. After applying n many actions on S_0 we obtain

$$S_0 \overset{\alpha_1,\theta_1}{\Longrightarrow} S_1 \overset{\alpha_2,\theta_2}{\Longrightarrow} \ldots \overset{\alpha_n,\theta_n}{\Longrightarrow} S_n.$$

A *planning task* $P := (S_0,\mathcal{A},G)$ consists of an initial state S_0, a goal G and a set of actions $\mathcal{A}$. The goal G is an existentially closed conjunction of atoms. If a sequence of descriptions $\alpha_1, \ldots, \alpha_n$ and a sequence of substitutions $\theta_1, \ldots, \theta_n$ exist, such that a grounded instance of G is true in S_n, then an n-plan exists, which is a plan of length n.

Example 2.2.2 (Formal definition of pegsol-invasion)**.** *Here we define a formal pegsol-invasion instance $P = (S_0,\mathcal{A},G)$ motivated by Example 2.2.1. The necessary FOL language $\mathcal{L}$ consists of variables X,Y,Z, the constants l_1,l_2,l_3,l_4 representing four locations, the 3-ary predicate "inline", and the unary-predicate "filled".*
The initial configuration states which locations are in line and initially filled.

$$\begin{aligned} S_0 = \quad \{ \quad & inline(l_1,l_2,l_3), inline(l_2,l_3,l_4), \\ & inline(l_3,l_2,l_1), inline(l_4,l_3,l_2), \\ & filled(l_1), filled(l_2)\} \end{aligned}$$

All grounded atoms which are not inside S_0 (e.g., $filled(l_3)$) are false in S_0. Thus, we know that l_3 is not filled (or free) in S_0.
The actions $\mathcal{A}$ consist only of one action which is called "jump" and defined as follows:

- $Name_{jump}$: $jump(X,Y,Z)$
- Pre_{jump}: $\{inline(X,Y,Z)$, $filled(X)$, $filled(Y)$, $\neg filled(Z)\}$
- Del_{jump}: $\{filled(X),filled(Y)\}$
- Add_{jump}: $\{filled(Z)\}$

The atoms $filled(l_i)$ with $i \in \{1,\ldots,4\}$ are fluents, because they are modified by action $jump(X,Y,Z)$. In contrast, $inline(l_x,l_y,l_z)$ with $x,y,z \in \{1,\ldots,4\}$ are not fluents.
Finally the goal is defined as $G = \{filled(l_3)\}$. A 1-plan exists with $\alpha_1 = jump$ and $\theta_1(X,Y,Z) \rightarrow (l_1,l_2,l_3)$.

Listing 2.1: Domain encoding of PDDL instance of pegsol-invasion

```
(define (domain pegsol-invasion)
    (:requirements :typing) (:types location - object)
    (:predicates
        (IN-LINE ?x ?y ?z - location)
        (filled ?l - location)
    )
    (:action jump
        :parameters
            (?from - location
            ?over - location
            ?to - location)
        :precondition
            (and
                (IN-LINE ?from ?over ?to)
                (filled ?from)
                (filled ?over)
                (not (filled ?to))
            )
        :effect
            (and
                (filled ?to)
                (not (filled ?from))
                (not (filled ?over))
            )
    )
)
```

PDDL

We use the PDDL definition for a planning task $P = (S_0,\mathcal{A},G)$ with $\mathcal{L}$ the language of P from [KBC+98]. The planning task is split into two files called `domain.pddl` and `problem.pddl`.
A domain file consists of `requirements`, `types`, `constants`, `predicates` and `actions`. An action is defined by its `parameters`, `preconditions` and `effects`.
A problem file consists of a `domain` reference, an `init` and a `goal` description.
Preconditions and goal descriptions are function-free first-order formulas. Additions and deletions are grouped into one place, the effects. Instead of a delete list, a *deletion* is achieved by mentioning the negation of a statement as an effect. This was not possible before because both the addition and the deletion lists contain only atoms. *Conditional effects* and *universal quantifiers* can be used while describing the effects. The requirements of a planning task state, what a planner needs (from a set of requirement flags) to handle in order to solve the planning task. This is necessary as some planners focus on less complex domains. E.g., if a planner uses the STRIPS-blueprint, it cannot handle conditional effects, which can be described in PDDL. However, the expressiveness of PDDL is restricted compared to the formal definition introduced before [KBC+98]. Every constant, predicate and action is defined respectively to the available types of objects.

Listing 2.2: Problem encoding of PDDL instance of pegsol-invasion

```
(define (problem pegsol-invasion-example)
    (:domain pegsol-invasion)
    (:objects
        l1 - location
        l2 - location
        l3 - location
        l4 - location)
    (:init
        (IN-LINE l1 l2 l3)
        (IN-LINE l3 l2 l1)
        (IN-LINE l2 l3 l4)
        (IN-LINE l4 l3 l2)
        (filled l1)
        (filled l2)
    )
    (:goal
        (filled l3)
    )
)
```

Similar to the SMT-LIB format, PDDL is a standard for planning tasks, making it easy to compare different tools.

Example 2.2.3 (PDDL definition of pegsol-invasion)**.** *A PDDL pegsol-invasion instance motivated by Example 2.2.1 with given planning task $P = (S_0,\mathcal{A},G)$ with language $\mathcal{L}$ can be found in Listings 2.1, 2.2. The requirements demand that typing is handled by the planner, in order to introduce the type location. The predicates are the same which appear in $\mathcal{L}$ and the action jump is defined similar to the jump action inside $\mathcal{A}$. The initial and goal configuration are defined respectively to S_0 and G from P.*

Planning as Satisfiability

In this section we present the *Planning as Satisfiability* paradigm [KSAH92], [KMS96]. Using a PDDL specification, combined with a plan horizon $ph \in \mathbb{N}$, it is possible to encode the dynamics into an SMT-LIB file. This file represents a FOL formula which can be transformed into a set of clauses φ^{ph}, which is solvable with SMT solvers described in Section 2.1.3. If the solver detects satisfiability, we know that the planning task is solvable and a ph-plan exists. Otherwise the solver detects unsatisfiability and we know that no ph-plan exists. Note that unsatisfiability of one encoding does not prove infeasibility of the planning task, although, as we will see in Chapter 3, it can yield a witness for infeasibility.
Various approaches exist to translate a PDDL planning task into an SMT problem, such as *SMTPlan* [CFLM16], *Reinforced Encoding* [BBT15], and one based on the SAS+ formalism [HCZ10]. The disadvantage of automatically created translations is their size, as they offer functionalities, we are not interested in (e.g., SMTPlan

enables the planning for hybrid systems). Therefore, we construct hand-crafted encodings, following the description in [Tac18] closely. The structure of the hand-crafted encodings includes variable declaration, bounds on variables, action definitions and initial/goal clauses (Example 2.2.4).

Definition 2.2.1 (SMT-LIB encoding φ^{ph})**.** *For a plan horizon ph and a planning task $P = (S_0,\mathcal{A},G)$ with language $\mathcal{L}$ we can define a FOL formula, transformable into a set of clauses φ^{ph}. Assume that $\mathcal{L}$ does not contain any function symbols (we will see later, this assumption rules out undecidable planning tasks).*

$$\varphi^{ph} := \texttt{getClauses}\,(\varphi^{ph}_{action} \wedge \varphi_{initial} \wedge \varphi^{ph}_{goal})$$

with `getClauses`(φ) *transforming a FOL formula φ into a equisatisfiable set of clauses.*
We introduce for every predicate symbol in $\mathcal{L}$ and plan step $1 \leq i \leq ph$ all possible variables. To do so, we need to list all grounded predicates. We add the action variables A_i for every plan step $1 \leq i \leq ph$, encoding which action is used in which step.
The formula φ^{ph}_{action} encodes all possible actions for every plan step by specifying their preconditions, effects and what does not change inside the domain for every plan step from 2 up to ph. The structure of the encoding of a single action a_i states, if the action identified by the index i is chosen, its preconditions, effects and conditions, what is not allowed to change, are enforced to be fulfilled.

$$\varphi_{action^j_i} := (A_j = i) \Rightarrow \left(\varphi^i_{preconditions} \wedge \varphi^i_{effects} \wedge \varphi^i_{notchange}\right)$$

The amount of possible actions N_A is obtained by listing all possible groundings of all actions in $\mathcal{A}$. Due to the fact, the amount of possible actions might be much greater than the amount of executable actions, we can reduce this number by additional domain knowledge and manually modifying the encoding. The action variables must be restricted to the size of possible actions, $(1 \leq A_j) \wedge (A_j \leq N_A)$.
The formula $\varphi_{initial}$ contains a full specification of the world state. Note that in the formal definition it is enough, to state what holds initially in S_0 and every grounded atom which is not contained, is assumed to be false in S_0. The formula φ^{ph}_{goal} mentions the variables corresponding to all grounded atoms in G.
We encode the initial configuration, the transitions between states and the goal configuration. If the encoding is satisfiable for some $ph \in \mathbb{N}$ we can analyze the assignment and print the action-indices assigned to the action variables in every step. Otherwise the encoding is unsatisfiable, stating that it is not possible to reach G from S_0 by applying ph many actions.

Example 2.2.4 (Planning as Satisfiability definition of pegsol-invasion)**.** *Here we define a pegsol-invasion instance referring to the Planning as Satisfiability paradigm motivated by Example 2.2.1 for a given planning task $P = (S_0,\mathcal{A},G)$ with language $\mathcal{L}$. We use φ^{ph} as a FOL formula over Boolean and Integer variables. The boolean variables, taken from $\mathcal{L}$, are $filled^j_i$ and $inline^j_{x,y,z}$ with $i,x,y,z \in \{1,\ldots,4\}$ representing the locations and $j \in \{1,\ldots,ph\}$ the plan step. The integer variables are A_j, representing which action is chosen at step $j \in \{1,\ldots,ph\}$ of the plan.*
As indicated, we can reduce the size of variables by only using $inline_{x,y,z}$ for one plan step instead of $inline^j_{x,y,z}$, because $inline_{x,y,z}$ does not represent a fluent and will not change throughout the plan.

The possible amount of actions depends on the domain and number of variables in this action. The jump action has three variables which represent all locations. Because we have 4 locations, we obtain $4{\cdot}3{\cdot}2 = 24$ *actions. If we replace the index of each action by the 3-tuple of corresponding locations we get a set of actions* $\mathcal{A} = \{a_{123}, a_{124}, \ldots, a_{432}\}$ *containing all 24 actions* a_{xyz}*. Each action indicates jumping from* x *over* y *to* z*. E.g., action* a_{123} *encodes jumping from* l_1 *over* l_2 *to* l_3*:*

$$\varphi^j_{action_{123}} := (A_j = 1) \Rightarrow \quad \overbrace{inline_{123} \wedge filled_1^{j-1} \wedge filled_2^{j-1} \wedge \neg filled_3^{j-1}}^{preconditions} \wedge \overbrace{\neg filled_1^j \wedge \neg filled_2^j \wedge filled_3^j}^{effects} \wedge \overbrace{filled_4^j = filled_4^{j-1}}^{notchange}$$

Note we assume, that action a_{123} *is the first element of the set of actions. The other actions are encoded in a similar way.*
The initial and goal configuration are defined as follows:

$$\begin{aligned} \varphi_{initial} &:= inline_{123} \wedge inline_{321} \wedge \\ &\quad inline_{234} \wedge inline_{432} \wedge \\ &\quad \neg inline_{124} \wedge \ldots \wedge \neg inline_{431} \wedge \\ &\quad filled_1^1 \wedge filled_2^1 \wedge \neg filled_3^1 \wedge \neg filled_4^1 \\ \varphi_{goal} &:= filled_3^{ph} \end{aligned}$$

Here we applied already the optimization that we are only using one plan step for every non-fluent variable.

Lemma 2.2.1. *The planning task* $P = (S_0, \mathcal{A}, G)$ *with language* $\mathcal{L}$ *is solvable iff the encoding* $\varphi^{ph_{min}}$ *is satisfiable for some minimal* $ph_{min} \in \mathbb{N}$*.*

Proof. $\Rightarrow$ We know that the given planning task is solvable, and thus there exists some minimal ph_{min}-plan for $ph_{min} \in \mathbb{N}$, allowing to reach G from S_0. By definition the encoding for $\varphi^{ph_{min}}$ is satisfiable and yields the ph_{min}-plan as a side effect of the satisfying assignment.
As we know that the ph_{min}-plan is minimal, no $ph^\dagger$-plan with $ph^\dagger < ph_{min}$ allows reaching G from S_0. Thus, by definition all encodings $\varphi^{ph^\dagger}$ are unsatisfiable.

$\Leftarrow$ Because φ^{ph} is satisfiable, we can extract a ph-plan from the satisfying assignment proving feasibility of the given planning task. □

Once we have the general idea in mind, we can combine the encodings with the principles of *Bounded Model Checking* (BMC) [Bie09]. At the beginning of BMC an upper bound b is defined. The highest bound is called the *completeness threshold* ($b \leq \mathcal{CT}$). For every given *finite state system* such an $\mathcal{CT}$ exists.
The loop starts with some value v stating that the system executes v many steps. If

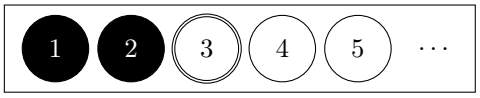


Figure 2.3: Undecidable pegsol-invasion example

the encoding for v is satisfiable the property has been proven, and if it is unsatisfiable v is increased up to b.
What can be ensured once the upper bound is reached? If the upper bound $\mathcal{CT}$ is reached the investigated property does not hold, otherwise we find a *witness* for some value $v < \mathcal{CT}$. *Classical planning* is only concerned with safety properties and thus less complex than the diverse properties investigated in *model checking* (as *liveness* and *nested temporal properties*). In more detail for classical planning, if the upper bound $\mathcal{CT}$ is reached without finding any plan yet, we conclude that no plan exists for the given problem; otherwise we find such a plan for some plan horizon $ph < \mathcal{CT}$. Later we will see that depending on the complexity of the planning domain the problem instance is *decidable* or not. This fact states, the given system does not represent a finite state system anymore. Therefore it cannot be ensured that a completeness threshold exists.

It is possible to use domain knowledge to pre-compute a fixed plan horizon ph_{fix} such that satisfiability of $\varphi^{ph_{fix}}$ ensures plan existence. Otherwise plan infeasibility is guaranteed. An example for this approach is an encoding for the *RoboCup Logistics League* (RCLL) [NRF+15]. The RCLL simulates an industry 4.0 environment by asking a robot fleet to work on dynamically announced orders. The robots have to coordinate their actions in order to gain as many points as possible in a fix time window. Depending on the complexity of orders we know which actions are required. Then only the best partial order and assignment of robots matters. However, it is difficult to argue, if those plan horizons exist for a given planning task without domain dependent knowledge. A possible approach to answer this question consists of an exhaustive state space search in order to analyze all possible plans.

2.2.2 Decidability

As indicated in the description of Planning as Satisfiability decidability of a planning task is an important property to detect its infeasibility. We refer to the definition by [ENS92]. Planning is in general undecidable, if allowance of function symbols is granted or an infinite number of constant symbols are available. If both properties do not hold the planning task is decidable.

Example 2.2.5 (Decidable planning task)**.** *The planning task instance specified in Example 2.2.1 is decidable as we have no function symbols and only finitely many constants. We obtain an undecidable instance by imagining that we have an infinite line of locations as displayed in Figure 2.3. This results in an infinite number of locations (constant symbols). Depending on the order in which we test the available actions we find the solution or not (making the process semi-decidable).*

2.2.3 Tools

Even though state-of-the-art planners focus on detecting plans for solvable planning tasks, some planners, which are dedicated to conclude infeasibility of planning tasks, exist as well.

Solvable Planning Tasks

For solving a planning task, planners aim, for example, to generate *optimal*, *bounded-cost*, *satisficing* or *agile* plans in the *International Planning Competition* at ICAPS 2018[2]. The first three variants aim to generate good plans with respect to their cost and the last variant focuses on the time for generating a single plan. Examples are *LAPKT-BFWS-Preference* [FGFR18], *Saarplan* [FGS18], and *Remix* [Sei18].
The tool *LAPKT-BFWS-Preference* has the best performance on the agile track. It is based on the *Best-First Width Search* (BFWS) approach. Due to the focus on *state novelty* it is *search-dependent* but *goal-independent* [KT17]. In general, the state novelty $\omega(s)$ of a state s is defined as the smallest set of atoms Q which appear first in some state. For the given tool this principle has been extended by a set of heuristic functions $h_1, \ldots, h_n$. Then we prune all states for which $\omega(s) > k$ with a threshold k. If k converges to the number of atoms in the planning task the algorithm simulates *Breath First Search* (BFS). We obtain polynomial complexity for fix values of k while omitting completeness. Surprisingly, the quadratic case of $k = 2$ solves already many planning tasks. The final version for the competition uses some post-processing to optimize a found plan with the approaches *WA** [RW14] and h^2 [AT15] up until the time window ends.

Unsolvable Planning Tasks

Recently the problem of unsolvable planning tasks has been evaluated in the *Unsolvability International Planning Competition* at ICAPS 2016[3]. Various approaches have been used, among them are, *Aidos* [SPS+16], *SymPA* [Tor16], and *CLone* [SH16]. Here we will present the tool *SymPA*, a tool with one of the best performances.
SymPA, based on the *Fast Downward planning system* [Hel06], uses h^2 to pre-process the planning task [AT15] and is a variation of *SymBA** [TLB16]. The approach, similar to SymBA*, consists of a bidirectional search in the original and abstract state spaces in order to find dead ends. Abstract state spaces describe planning tasks induced by a subset of the variables in the original planning task. They are closely related to the term *subtask* introduced in Section 2.2.4.

2.2.4 Further Definitions

To improve readability throughout this thesis we present some further definitions concerning planning tasks.

Encoding φ^{ph}

Each encoding of the set of clauses φ^{ph} of the planning task for a plan horizon ph contains a subset of clauses $\varphi_{initial}$ and φ^{ph}_{goal} representing the initial and goal configuration of the planning task. The remaining information is described by the set of clauses φ^{ph}_{domain}.

$$\varphi^{ph} := \varphi_{initial} \cup \varphi^{ph}_{goal} \cup \varphi^{ph}_{domain}$$

Plan Horizon *ph*

In general, we will describe our approach referring to the current *plan horizon*. This plan horizon represents the current bound in the BMC paradigm. If we are given a fixed plan horizon ph we identify all smaller plan horizons as $ph^{\dagger}$ and all greater once as ph^{*} with $ph^{\dagger} < ph < ph^{*}$, unless specified otherwise.

Subtask of a Planning Task

We define a subtask as an abstraction of a given planning task by relaxing certain properties of the initial or goal configuration.

Definition 2.2.2 (Subtask S)**.** *A subtask S of the original planning task $P = (S_0,\mathcal{A},G)$ with language $\mathcal{L}$ and the corresponding set of clauses φ^{ph} for some $ph \in \mathbb{N}$ is a subset of the clauses $\varphi^{ph}_S \subset \varphi^{ph}$ with*

- $\emptyset \neq \varphi_{S,initial} \cup \varphi^{ph}_{S,goal} \subset \varphi_{initial} \cup \varphi^{ph}_{goal}$,
- $\varphi^{ph}_{S,domain} = \varphi^{ph}_{domain}$.

We call a subtask trivial if either no initial or no goal clauses appear. All other subtasks are non-trivial.

In terms of the formal definition of planning tasks, a subtask can be understood as a set of planning tasks, where all dropped initial or goal configurations appear with all their possible variations.

Example 2.2.6 (Subtask of pegsol-invasion)**.** *A planning task of pegsol-invasion consists of the operator definition and an initial and goal configuration. The initial configuration states, for example, whether a location is filled or not. Let us now consider the subtask where the information that l_4 is initially free is dropped. This results in two new planning tasks. In one l_4 is still free, but in the other l_4 is considered filled. If we recall Example 2.2.4 the subtask would correspond to the initial specification $\varphi_{S,initial} := \neg free^1_1 \cup \neg free^1_2 \cup free^1_3$. The variable $free^1_4$ is not fixed by $\varphi_{S,initial}$ anymore.*
This subtask is non-trivial as the information about fluent $free^{ph}_3$ is contained in the goal configuration, besides other specifications in the initial configuration.

Definition 2.2.3 (Feasibility of subtasks)**.** *A subtask is solvable iff one of its planning tasks is solvable. The subtask is unsolvable iff no planning task is solvable.*

Definition 2.2.4 (Minimal unsolvable and maximal solvable subtask)**.** *An unsolvable subtask is minimal iff every further subtask is solvable. Contrary, a solvable subtask is maximal iff every supertask of the subtask is unsolvable.*

If a planning task consists of several subtasks, plan infeasibility of one subtask is a witness for infeasibility of the original planning task. The other way does not hold. It is not true that if a planning task is unsolvable, all subtasks are unsolvable as well. A *trivial* subtask shares no fluents between its initial and goal state and thus feasibility does not depend on any actions as all fluents in the goal configuration can be assigned with their initial value. Later in Section 3.1 we will see that minimal subtasks are closely related to MUSs in the context of Planning.

2.3 Explainable Artificial Intelligence

This section offers an overview about challenges in *eXplainable AI* (XAI) in general and *Explainable Planning* (XAIP) in particular. Recent development in this field of research has been pushed by DARPA's XAI program[9]. The program aims in creating AI systems and models which can explain themselves.

A popular example of such a self-explaining system is the *decision tree* (DT) [Qui86]. A DT aims to classify objects specified by a set of attributes. The procedure a DT is based on is easily accessible to humans and therefor fulfills the requirements asked for by the program. An explanation is simply that the values of the attributes conclude a certain class. However, a DT is limited in its expressiveness and easily outperformed by tools with a similar functionality.

Let us consider a contrary example, which is not self-explanatory, from some computer vision system. The task is that after looking at a picture the system declares whether it displays a dog or not[1]. After the system is trained with some data set, it can be applied in daily life. The decision the classifier makes can be correct or wrong. How can the system explain its conclusions? Of course, it is possible to simply provide the principles (via a paper-reference) or the implementation (via the source-code) of the tool. This can be difficult to understand for a user with no technical background. A more convenient way could be to highlight a segment of the image, which indicates a dog appears on the picture. An alternative is to refer to the head, the tail, the body, and the legs. These properties indicate that it is, for example, an animal. Further, the shape of the ears refers to a dog. The highlighted visual information is easily interpretable by humans, while the theoretical principles or the source-code are not.

In general, it can be observed, the more powerful an AI system is, the less understandable it is by humans. Such complex programs are often regarded as *black-boxes* stating that no effort is made in explaining any component[9]. A problematic consequence of such black-boxes arises in crucial situations where all possible actions have far-reaching effects. If the human's intuition contradicts the decision of the system, the human will be left confused and follow some intuition accompanied with one of these thoughts: 1. *»Who knows how the system concluded that action? I will perform an action based on my experience!«* or 2. *»Even if I do not understand the system it will surely have its reasons. I rather disregard my experience!«*
Both possibilities are not what is intended by the use of computer systems and can be ameliorated by XAI such that the following scenarios appear: 1. *»The machine put too much attention on less important details. I will perform an action based on my experience!«* or 2. *» I accept that I forgot important details. I rather disregard my experience!«*
[FLM17] states that basically three reasons form the basis of XAI: 1. Trust, 2. interaction, and 3. transparency. Trust is needed in cases that computer systems assist humans. Interaction is required if full autonomy is not possible and humans and computer systems operate side by side. Transparency is necessary in full autonomic applications, e.g., for legal issues, such that performed actions can be ensured not to be discriminating.

[9] https://www.darpa.mil/program/explainable-artificial-intelligence

Algorithm 6 CAMUS with clauses of interest

```
1: procedure CAMUS(φ_i,φ_n)
2:     φ_selector ← getSelector(φ_i)
3:     φ_BCs = ∅
4:     k = 0
5:     while solve(φ_selector ∪ φ_n ∪ φ_BCs) do
6:         k++
7:         φ_atmost_k = φ_selector ∪ φ_n ∪ atmost(k)
8:         while solve(φ_atmost_k ∪ φ_BCs) do
9:             Model m ← getModel(φ_atmost_k ∪ φ_BCs)
10:            φ_BCs = φ_BCs ∪ getBlockingClause(M)
11:    return hittingSetGeneration(getMCSs(φ_BCs))
```

2.3.1 Explainable Planning

In this thesis we focus on the problem of XAIP. Several topics are analyzed in this field of research. *Plan Explanation* aims to present generated plans in an understandable form [SBM11], [CFT+18]. *Plan Explicability* analyzes how well generated plans can be understood by humans [ZSK+17]. Based on these approaches, *Model Reconciliation* reasons about plans consistent with the human's view on the domain [CSK17]. Moreover, efforts have been made to provide decision support in the context of *Human Team Planning* [KS17]. In this context several questions are of interest [FLM17]. For simplicity the proposed questions are grouped into the following:

1. *Why did you do that?*
2. *Why do I need to replan at this point?*
3. *Why can't you do that?*

The first question aims to explain the necessity of some actions with respect to similar actions and overall cost by executing them. Answers are that discarding some actions results in a plan with worse metric or could even prohibit plan existence.
The second question is out of scope for this thesis, as it requires monitoring and replanning while the initial plan is currently executed.
The third question demands an explanation for infeasibility. A planning task can be impossible to solve due to the 1. domain itself, 2. the environmental conditions and/or 3. the goals one tries to achieve. Note that several causes can exist prohibiting feasibility of a planning task.

Example 2.3.1 (Passing a bridge)**.** *Imagine that we try to reach a destination and need to pass two bridges. If the first bridge is already down, we cannot reach the destination, independent of the second bridge. But if the second bridge is down as well it does not matter if the first bridge is up again. Returning only the information that the first bridge is down is enough to explain infeasibility itself, but for a wider picture it is helpful to understand all components causing infeasibility.*

In the next Chapter 3 we propose a technique which aims to detect causes of all three types regarding Question 3. and return this information in a useful format to the user.

Algorithm 7 Basic Linear Search (BLS) with clauses of interest

```
1:  procedure BLS(φ_i, φ_n)
2:      φ_BCs = ∅
3:      while solve(φ_BCs ∪ φ_n) do
4:          φ_MCS = ∅
5:          φ_MSS = φ_BCs ∪ φ_n
6:          for all clause ψ ∈ φ_i do
7:              if solve(φ_MSS ∪ ψ) then
8:                  φ_MSS = φ_MSS ∪ ψ
9:              else
10:                 φ_MCS = φ_MCS ∪ ψ
11:         φ_BCs = φ_BCs ∪ getBlockingClause(φ_MCS)
12:     return hittingSetGeneration(getMCSs(φ_BCs))
```

2.3.2 Clauses of Interest in Planning

After presenting approaches about how planning tasks are encoded as SMT problems and what is required in order to explain unsolvable planning tasks, we define an extension of the MUS generation algorithms allowing to define a subset of so-called *clauses of interest*. Instead of considering all clauses of a formula φ we define a classification into

$$\varphi = \varphi_{interest} \cup \varphi_{nointerest} = \varphi_i \cup \varphi_n.$$

The motivation of this procedure is, we might not be interested, if certain parts of the encoding contribute to MUSs or not. For example, the constraints bounding the action variable will appear in some MUS, because removing this restriction allows the solver to *define* unintended actions based on no rules. Each clause can be marked as a clause of interest or not. We make the choice from a computational point of view to only mark the initial and goal clauses as clauses of interest.

Both algorithms need to be modified in order to work with clauses of interest. In CAMUS we extend only the clauses of interest with clause selector variables and add φ_n to the two solver calls. In BLS, similarly, we add φ_n to the outer solver call. Moreover, we initialize the current MSS with the blocking clauses and φ_n. Finally, we do not iterate over all clauses but over all clauses of interest. In both algorithms the blocking clauses are only created by use of the clauses of interest. Changes can be seen in Algorithms 6, 7.
The returned MUSs consist of φ_n and a subset of the clauses of interest. In fact, they are not strictly minimal anymore, because it is possible that removing a clause from φ_n still yields a unsatisfiable set of clauses. Therefore, we say that the MUSs are *minimal with respect to the clauses of interest*. Removing any clause of interest from the obtained MUSs returns a satisfiable set of clauses. Moreover, we say that an MUS is *non-empty* with respect to the clauses of interest, if at least one of the clauses of interest appears in the MUS. Every time we use the expression MUSs from this point on, we refer to MUSs with respect to their clauses of interest.

Chapter 3

Deciding Planning Tasks with UNSAT Cores

In this chapter, we study MUSs with respect to clauses of interest in the context of Planning and present an approach on how to use those together with the original BMC approach which we call ***B**MC using **UNS**AT cores* (BUS). First, we classify UNSAT cores into three distinct classes and analyze their impact on plan feasibility. Section 3.2 offers a description of BUS and its correctness is examined in Section 3.2.2. Further comments on the complexity of BUS are presented in Section 3.2.3.

Besides the condition that given planning tasks must be decidable we put an additional requirement on the encoder translating PDDL specifications into SMT encodings. The plans returned in the solving process are not allowed to contain any loops. This requirement does not reduce the expressiveness of the planning problem, as a loop does not add any information to the plan. Omitting loops in the satisfying assignments in our encodings is crucial for BUS (Proof 3.2.2). To do so we have to add the following formula for a set $\mathcal{F}$ of fluents and the plan horizon ph:

$$\varphi^{ph}_{omitloops} := \bigwedge_{i<j\leq ph} \bigvee_{f\in\mathcal{F}} f^i \neq f^j$$

with f^i the variable encoding the assignment of fluent f in plan step i and i,$j \leq ph$. The formula $\varphi^{ph}_{omitloops}$ ensures that at least one pair of assignments for each pair of plan steps (i,j) with $i < j$ differs. Once $\varphi^{ph}_{omitloops}$ is transformed into an equisatisfiable CNF-formula we add its set of clauses to the current encoding φ^{ph} of the planning task.

3.1 Classification of UNSAT Cores

Motivated by Section 2.3.2, the clauses in the encoding φ^{ph} of the planning task can be classified into the initial and goal configurations as clauses of interest and the domain as the other clauses.

$$\begin{aligned}\varphi^{ph} &= \varphi_i^{ph} \cup \varphi_n^{ph}\\ \varphi_i^{ph} &= \varphi_{initial} \cup \varphi_{goal}^{ph}\\ \varphi_n^{ph} &= \varphi_{domain}^{ph}\end{aligned}$$

with $(\varphi_{initial} \cup \varphi_{goal}^{ph}) \cap \varphi_{domain}^{ph} = \emptyset$, and $\varphi_{initial} \cap \varphi_{goal}^{ph} = \emptyset$. Throughout enrolling the transition sequence we can apply CAMUS or BLS for every plan horizon $ph \in \mathbb{N}$ to generate all MUSs $C_1, \ldots, C_N$. These MUSs are of the following form:

$$\varphi_{C_i}^{ph} = \varphi_{C_i,initial} \cup \varphi_{C_i,goal}^{ph} \cup \varphi_{C_i,domain}^{ph}$$

with $i \in \{1, \ldots, N\}$, $\varphi_{C_i,initial} \cup \varphi_{C_i,goal}^{ph} \subseteq \varphi_{initial} \cup \varphi_{goal}^{ph}$ and $\varphi_{C_i,domain}^{ph} = \varphi_{domain}^{ph}$. Because the domain clauses are not among the clauses of interest they do not change for any MUS and do not need to be analyzed further. The set of clauses of interest appearing in an MUS C_i is a subset of the original clauses of interest. We use those clauses to classify possible UNSAT cores in order to argue about their impact on XAIP and plan feasibility.

Definition 3.1.1 (Classes of UNSAT cores with respect to clauses of interest in the context of Planning)**.** *We propose three classes of non-empty UNSAT cores. If the UNSAT core generation algorithm only generates the empty MUS, we call this the empty core. Otherwise we obtain a non-empty MUS C.*

- If $\varphi_{C,goal}^{ph}$ is empty, we call C an *initial core.*
- Else if $\varphi_{C,initial}^{ph}$ is empty, we call C a *goal core.*
- Else we call C a *mixed core.*

Example 3.1.1 (An initial, goal and mixed core in the pegsol-invasion planning task)**.** *We refer to Example 2.2.4. The UNSAT core C_1 with the clauses of interest*

$$\varphi_{C_1,initial} \cup \varphi_{C_1,goal}^{ph} := \{\neg free_1^1, \neg free_2^1\}$$

is an initial core. The UNSAT core C_2 with the clause of interest

$$\varphi_{C_2,initial} \cup \varphi_{goal}^{ph} := \{\neg free_3^{ph}\}$$

is a goal core. The combination of the above UNSAT cores resulting in C_3 with the clauses of interest

$$\varphi_{C_3,initial} \cup \varphi_{C_3,goal}^{ph} := \{\neg free_1^1, \neg free_2^1, free_3^{ph}\}$$

is a mixed core.

As indicated before in Section 2.2.4, an MUS defines a subtask because removing initial and goal clauses allows the solver to search for a *ph*-plan solving the subtask by re-assigning variables, which have been assigned via the initial and goal clauses before. In fact, a mixed core describes a non-trivial subtask and an initial or goal core describes a trivial subtask. Thus, every MUS states that no *ph*-plan solves a certain subtask.

Lemma 3.1.1. *An MUS C in the context of Planning appears for some plan horizon ph iff no ph-plan exists solving the minimal subtask S_C defined by C.*

Proof. $\Rightarrow$ Assume that the subtask defined by an MUS is solvable by a ph-plan. Then the encoding of the subtask for plan horizon ph is satisfiable. This contradicts the assumption because a UNSAT core is unsatisfiable by definition.
Now we investigate the case that no ph-plan solves S_C but that S_C is not minimal and some subtask of S_C exists which is not solvable by any ph-plan as well. In this case C is no true MUS, as the subtask of S_C states, there is a smaller unsolvable subtask than S_C. This contradicts that C appeared.

$\Leftarrow$ If no ph-plan exists solving the minimal subtask S_C, an encoding exists corresponding to the subtask which is unsatisfiable. Because the subtask is minimal, every subtask of S_C can be solved by a ph-plan, and therefore the encoding truly defines an MUS. □

In case of a trivial subtask, the initial or goal core do not only state that no ph-plan exists, but no ph^*-plan as well.

Lemma 3.1.2. *If the empty core, or an initial or goal core C appears for some plan horizon ph then no ph^*-plan exists solving the minimal subtask S_C defined by C for $ph \leq ph^*$*

Proof. We consider the case that an initial core appeared. Assume that the subtask defined by an initial core is solvable by some ph^*-plan. Then we can create a ph-plan by removing the last $(ph^* - ph)$ steps of the ph^*-plan. This contradicts the fact that the appearance of an initial core prohibits existence of a ph-plan after Lemma 3.1.1. The case of the empty and a goal core works similarly. □

Note that the other direction does not hold, because unsolvable non-trivial minimal subtasks exist as well.

3.1.1 Impact of UNSAT Cores

Each introduced class has influence on plan feasibility. We investigate for each class what we can deduct from its appearance after generating all MUSs for some plan horizon ph using Lemmas 3.1.1, 3.1.2.

- The empty core states that no ph-plan exists in the given domain, because the encoding of φ^{ph}_{domain} is already unsatisfiable. Intuitively, even though we can specify any planning tasks by setting the initial and goal configuration it is impossible to obtain any executable ph-plan. This is a strong argument as it states that ph is an upper bound for all possible plans in the domain. Sound plans can only be of length $ph^\dagger$. Therefore, the empty core is a witness for plan infeasibility if no $ph^\dagger$-plan has been found solving the original planning task.

- An initial core states that no ph^*-plan with $ph^* \geq ph$ exists for the corresponding trivial subtask as shown in Lemma 3.1.2. Thus, no sequence of states exists starting in S_0 and reaching any S_{ph} and only dead ends have been found in the forward search. Therefore an initial core is a witness for plan infeasibility if no $ph^\dagger$-plan has been found solving the original planning task.

- A goal core states that no ph^*-plan with $ph^* \geq ph$ exists for the corresponding trivial subtask as shown in Lemma 3.1.2. Thus, no sequence of states exists starting in some S'_0 and reaching a grounded instance of $G = S_{ph}$ and only dead ends have been found in the backward search. Therefore a goal core is a witness for plan infeasibility if no $ph^\dagger$-plan has been found solving the original planning task.
- A mixed core states that no ph-plan exists for the corresponding non-trivial subtask, but some ph-plan do exist which solve subtasks of this subtask. Thus, extensions of the existing ph-plans result in dead ends or in a ph^*-plan solving the subtask (or even the original planning task).

The information we get from the MUSs is similar to reachability information collected in tools as *iProverPlan* [SK16] and *SymPA* [Tor16]. However, the way this information is collected is different. Instead of maintaining a set of reachable states we make use of a principle mentioned in [Str04]. While enrolling the transition scheme of the planning task, a conflict can occur stating that no connection exists between an initial and goal configuration. This information is detected by the use of MUSs and the appearing of the empty core, or initial or goal cores.

It is possible to use less clauses of interest as described so far. In general, less clauses of interest speed up the UNSAT core generation algorithms and are more accessible. However, depending on the initial or goal clauses we do not mark as clauses of interest, we have to redefine our classification. For example, if we do not consider some initial clauses as clauses of interest, those clauses will appear in every MUS. Therefore, it is impossible to detect the empty core or a goal core. If we detect the empty core with the modified clauses of interest the empty core corresponds to an initial core, and a goal core corresponds to a mixed core. Only the initial core is a correct initial core. The empty core and an initial core are still witnesses for plan infeasibility.

3.2 Bounded Model Checking with UNSAT Cores

In this section we describe how ***B**MC using **UNS**AT cores* (BUS) works in the context of Planning. The classical approach of BMC (Section 2.2.1) uses some completeness threshold $\mathcal{CT}$ up to which we increase our bound. In practice it is not applicable to use $\mathcal{CT}$ as an upper bound. Rather we use a bound of which we know we can reach it in a reasonable time. But in this case reaching the bound does not answer if the planning task is unsolvable. By use of the BMC approach we have three possible answers in terms of XAI: The planning task is 1. solvable, 2. unsolvable or 3. it might be solvable.
We use the interpretations of classes in Section 3.1.1 as a motivation to decide for each plan horizon, whether to increase the plan horizon or deduce that the given planning task is unsolvable. A visualization of the approach can be found in Figure 3.1.
The difference between BUS and the classical BMC approach is the moment we can deduce plan infeasibility. Originally, plan infeasibility can only be stated once $\mathcal{CT}$ is reached and no plan was found. If $\mathcal{CT}$ is not reached, we can only return the third answer. In contrast, BUS enables to return the second answer possibly for any plan horizon (in the worst case after reaching $\mathcal{CT}/2$ as we will see later in Proof 3.2.3). Note that depending on the planning task and the resources BUS does not find a witness for infeasibility and needs to state that the planning task might be solvable.

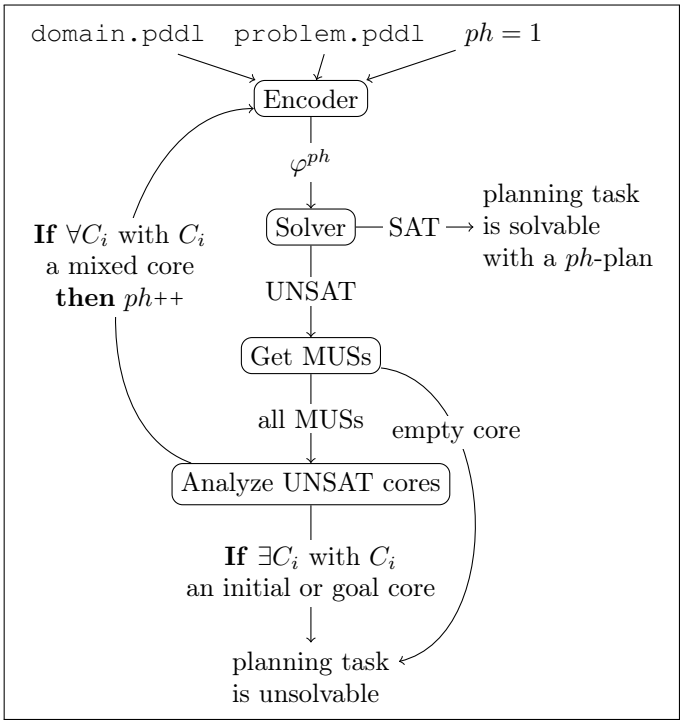


Figure 3.1: Visualized BUS for some planning task

3.2.1 BUS

We start with a planning task specified by its PDDL description The planning task's domain (`domain.pddl`) and problem description (`problem.pddl`), together with an initial plan horizon $ph = 1$, are then fed into an encoder producing an encoding representing a set of clauses φ^{ph}. Next a SMT solver is called with φ^{ph} as input.
If the formula is SAT, we abort the BMC procedure and return the ph-plan offered as a side product by the satisfying assignment. Otherwise we continue by generating all MUSs $C_1, \ldots, C_N$.
If the MUS generation yields only the empty core, we abort the procedure and return plan infeasibility. Else if one MUS is an initial or a goal core the planning task is unsolvable as well and we abort the procedure. Else all MUSs are mixed cores, stating that the planning task can be still solvable. Therefore, we increment ph by one and start all over with a new encoding.

Example 3.2.1 (BUS on an unsolvable instance of pegsol-invasion)**.** *In order to demonstrate BUS, we consider an unsolvable instance of the pegsol-invasion domain. The Example 2.2.1 is solvable but can be turned unsolvable by changing the goal configuration. Now we want location l_4 to be filled. There exist $2^4 = 16$ possible states as each location can be filled or free. Thus, we know that the longest possible plan has 16 steps without containing any loop (independent of the fact whether such a plan is sound) and a standard BMC procedure needs to enroll the transition scheme up to the completeness threshold of at most 16 without analyzing the domain further. The clauses of interest are:*

plan horizon	MCSs	MUSs
1	{3},{4},{5}	{3,4,5}
2	{4,5}	{4},{5}

Table 3.1: MCSs and MUSs generated throughout performing BUS on an unsolvable pegsol-invasion instance. The numbers indicate which clause appeared specified in Example 3.2.1.

1. l_1-*initial with* $filled_1^1$
2. l_2-*initial with* $filled_2^1$
3. l_3-*initial with* $\neg filled_3^1$
4. l_4-*initial with* $\neg filled_4^1$
5. l_4-*goal with* $filled_4^{ph}$

After running BUS on this instance, we obtain several MCSs and MUSs displayed in Table 3.1. The first iteration returns three MCSs containing each one clause, resulting into a mixed core with three clauses. Therefore, BUS decides to increase the plan horizon and generates one MCS with two clauses. This MCS yields two MUSs, one initial and one goal core. Each of these cores is a witness for plan infeasibility and causes BUS to stop, stating that the planning task is unsolvable.

3.2.2 Correctness

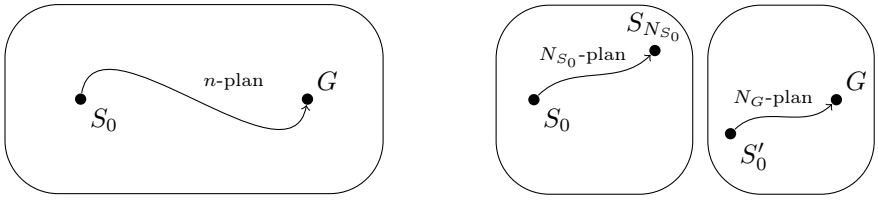
First, we proof the assumption that, in order for BUS to detect plan infeasibility, the encoding of a planning task has to prohibit loops in all plans.

Lemma 3.2.1. *If we can leave the initial respectively reach the goal configuration in up to ph steps without applying loops and a loop is possible on this ph-plan then no initial respectively goal core appears for any* ph^*.

Proof. Assume that an initial or goal core appears for $ph+1$, stating that no ph^*-plan exists (as shown in Lemma 3.1.2) and for all $ph^\dagger \leq ph$ only mixed cores appeared. We know that a loop is possible at some point in the ph-plan. Thus, we can construct a new ph^*-plan for $ph^* > ph$ where we run into and stay, or start in and leave the loop. Because ph^*-plans are possible, and therefore a $ph+1$-plan as well, no initial or good core can appear, contradicting our assumption. □

In order to prove correctness of BUS we have to prove the following theorem stating the relation between the original decidable planning task and BUS. BUS behaves correctly for solvable planning tasks as shown in Lemma 2.2.1.

Theorem 3.2.2. *The planning task is unsolvable iff the MUS generation returns only the empty core, or at least one initial or goal core on the encoding for some* $ph \in \mathbb{N}$ *and for every* $ph^\dagger < ph$ *the encoding is unsatisfiable and all cores are mixed cores.*



(a) Solvable planning task (b) Unsolvable planning task

Figure 3.2: The displayed paths in the state space indicate (a) a solving n-plan and (b) the longest possible plans from S_0 and to G.

Proof. $\Leftarrow$ Knowing that for ph and all $ph^\dagger < ph$ the encodings are unsatisfiable we have only to show that the same holds for all ph^* in order to prove infeasibility of the given planning task. If the empty core, or an initial or a goal core is returned no ph^*-plans exists (Lemma 3.1.2).

$\Rightarrow$ We know that the given planning task is unsolvable, and thus $\forall ph \in \mathbb{N}$ there exists no ph-plan allowing to reach G from S_0. Therefore, $\forall ph$ the corresponding encoding φ^{ph} is unsatisfiable and we can generate all MUSs. We show that if the planning task is unsolvable, then for some plan horizon ph the UNSAT core generation will return the empty core, or an initial or goal and all encodings $\varphi^{ph^\dagger}$ yield only mixed cores.
Because we cover decidable planning tasks, we know that there exists an upper bound N for all possible states in the domain. Moreover, because the given planning task is unsolvable, we can define two subsets of states. First the states reachable from S_0 and the states from which G is reachable (note that these two set of states do not strictly fill the complete state space as states might exist which are not reachable from S_0 or from which G is reachable). Let us define the longest plan executable from S_0 as the N_{S_0}-plan with N_{S_0} steps. Similarly, the longest plan executable to G is the N_G-plan with N_G steps. It is possible to extract trivial minimal subtasks S_{S_0} and S_G which still hold the same property. Because the original planning task is unsolvable, we know that no states reachable in N_{S_0} steps are among the states from which we can reach the goal state in less than N_G steps. These circumstances are displayed in Figures 3.2a, 3.2b. Even though the possibility of a N-plan exists, we can define the N'-plan as the longest possible plan in the domain with $N' \leq N$.
Assume that $N_{S_0} < N_G$ holds. We know that no $(N_{S_0}+1)$-plan exist for the trivial minimal subtask S_{S_0}. Once we reach the plan horizon $ph = N_{S_0}+1$ we obtain the corresponding initial core. The case of $N_{S_0} > N_G$ works in a similar way. The third case of $N_{S_0} = N_G$ and N_{S_0},$N_G < N'$ simply yields an initial and a goal core.
If $N_{S_0} = N_G = N'$ holds, then the MUS generation will return the empty core for the plan horizon $N'+1$. This is true, because due to the definition of N' no $N'+1$-plan exists in the domain.
For all $ph^\dagger$ only mixed cores appear. Assume that an initial or goal core appears for some $ph^\dagger$. Then we know that no $ph^\dagger$-plan exists for the corresponding initial or goal configuration. But such a plan can be simply constructed by omitting $(N_{S_0} - ph^\dagger)$ or $(N_G - ph^\dagger)$ actions from the available longest plans. □

3.2.3 Complexity

In this section we investigate the overall complexity in terms of how many iterations BUS requires. The advantage of using MUSs with BUS is a speedup in terms of solving less iterations. It is possible to answer the question whether a planning task is solvable or not within $\lceil \frac{\mathcal{CT}}{2} \rceil$ steps.
Moreover, we can use the MUSs to solve solvable subtasks and combine the generated plans to a plan solving the original planning task by matching the reached and left world state. Detecting a n-plan solving the planning task is possible in $\lceil \frac{n}{2} \rceil$ iterations. However, depending on the found solutions of the subtasks no match might be found. Note that in practice it is difficult to determine $\mathcal{CT}$. Even if we know $\mathcal{CT}$ it might be not possible to compute BUS up till $\lceil \frac{\mathcal{CT}}{2} \rceil$ from a computational point of view.

Lemma 3.2.3. *We need at most $\lceil \frac{\mathcal{CT}}{2} \rceil$ many iterations to terminate BUS for some planning task.*

Proof. We know that $\mathcal{CT}$ is the upper bound for the classical BMC procedure for some planning task. The bound $\mathcal{CT}$ stands for a $\mathcal{CT}$-plan, which is the longest possible plan in this domain. As indicated in Section 3.1.1 the MUSs tell us something about the forward and the backward search. If an initial core appears, we know that it is impossible to leave the initial configuration for some amount of steps, and correspondingly for goal cores. BUS will detect the smaller value first (or both at the same time if they are equal).
Knowing that the longest possible plan is a $\mathcal{CT}$-plan, we know that the worst-case detection of plan infeasibility is that we can move as many steps from some initial configuration as we can move to some goal condition. The longest possible *leave-* and *reach-plans* are $\frac{\mathcal{CT}}{2}$ (or $\lfloor \frac{\mathcal{CT}}{2} \rfloor$ and $\lceil \frac{\mathcal{CT}}{2} \rceil$ in the odd case) many steps long.
If no empty, initial or goal core appeared up to $\lceil \frac{\mathcal{CT}}{2} \rceil$ we know that it is possible to leave and reach the planning task for $\lceil \frac{\mathcal{CT}}{2} \rceil$ steps and therefore they must at least share one state S. The new plan to and from S is a witness for feasibility. □

Chapter 4

Explaining Planning Tasks with UNSAT Cores

In this chapter we present ideas how to use the MCSs and MUSs generated while performing BUS to explain the causes of infeasibility. More precisely we investigate what is not possible of a planning task (Section 4.1), what is possible (Section 4.2), and why some subtasks are not possible (Section 4.3). All presented features are implemented in the tool called `PEX` (*Planning EXplainer*).

As presented in Chapter 3 we obtain for every plan horizon in the procedure a set of MCSs and MUSs. Every iteration indicates solvable and unsolvable subtasks. Two useful possibilities exist among which the user can chose from.

1. Explain an unsolvable planning task with only one unsolvable subtask as a witness and collect all solvable subtasks on the way.

2. Explain an unsolvable planning task with all unsolvable subtasks as witnesses and collect all possible solvable subtasks. This requires an exhaustive search of the state space. Due to the possibly higher execution time we output available explanations on the run and provide the possibility to abort.

Additionally, to running BUS with the pegsol-invasion instance in Example 3.2.1 we introduce a new domain, chessboard-pebbling, in Example 4.0.1, which highlights other information which can be extracted from the generated MCSs and MUSs. The output of BUS on an unsolvable 3×3 chessboard-pebbling instance can be found in Example 4.0.2.

Example 4.0.1 (Chessboard-Pebbling)**.** *Chessboard-pebbling is one of the domains which we later use in the evaluation in Chapter 5. The task is to empty the top left location and its right and below neighbors of a two dimensional chess field. It is possible to modify the locations by the following rule: If a location is filled and the locations right and below are free, the location can be cleared by filling the locations right and below. The standard definition is unsolvable and can turned solvable by, for example, modifying the initially free locations. A visualization of a 3×3 chessboard-pebbling instance can be found in Figure 4.1.*

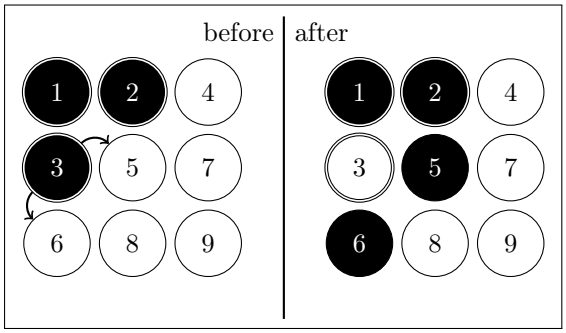


Figure 4.1: Visualized 3×3 chessboard-pebbling instance. Positions which need to be cleared are marked with a second circle. Initially filled locations are black.

Example 4.0.2 (BUS on an unsolvable instance of chessboard-pebbling)**.** *We use the unsolvable 3×3 chessboard-pebbling instance specified in Example 4.0.1. There exist* $2^9 = 512$ *possible states as each location can be filled or free. Thus, we know that the longest possible plan has 512 steps without containing any loop (independent of the fact whether such a plan is sound) and a standard BMC procedure needs to enroll the transition scheme up to the completeness threshold of at most 512 without analyzing the domain further. The clauses of interest are:*

1. l_1*-initial with* $filled_1^1$
2. l_2*-initial with* $filled_2^1$
3. l_3*-initial with* $filled_3^1$
4. l_1*-goal with* $\neg filled_1^{ph}$
5. l_2*-goal with* $\neg filled_2^{ph}$
6. l_3*-goal with* $\neg filled_3^{ph}$

Note that we did not marked the initial assignment of the other (all free) locations. This accelerates BUS on this instance and improves readability for the example. It can be argued if not marking those clauses as clauses of interest is a loss of information which could be used for explaining this planning task.
After running BUS on this instance, we obtain several MCSs and MUSs displayed in Table 4.1. The first four iterations yield only mixed cores. Then, the fifth iteration returns the empty core (actually an initial core as indicated in Section 3.1.1) and BUS terminates.

4.1 Unsolvable Subtasks

Each MUS indicates a subtask which is already unsolvable. All of them can be returned to the user to highlight difficulties of a planning task. This is valuable information for the user, even if we cannot gather enough information to state plan infeasibility due to computational limits. Unsolvable subtasks describe which components of the domain influence each other. Maybe the user has the power to modify

plan horizon	MCSs	MUSs
1	{1,2},{1,3},{1,5},{1,6}, {2,4},{2,3,5,6},{3,4}, {4,5},{4,6}	{1,2,4},{1,3,4},{1,4,5}, {1,4,6},{2,3,5,6}
2	{1,2},{1,3},{1,5},{1,6}, {2,4},{2,3,5},{2,3,6}, {2,5,6},{3,4},{3,5,6}, {4,5},{4,6}	{1,2,3,4},{1,2,4,5}, {1,2,4,6},{1,3,4,5}, {1,3,4,6},{1,4,5,6}, {2,3,5,6}
3	{1},{2,3,5},{2,3,6}, {2,5,6},{3,5,6},{4}	{1,2,3,4},{1,2,4,5}, {1,2,4,6},{1,3,4,5}, {1,3,4,6},{1,4,5,6}
4	{2,3},{2,5}, {2,6},{5,6}	{2,5},{3,6}
5	-	-

Table 4.1: MCSs and MUSs generated throughout performing BUS on an unsolvable chessboard-pebbling instance. The numbers indicate which clauses appeared specified in Example 4.0.2.

the domain (e.g., by replacing a battery) or to choose a different opportunity to tackle the planning task again (e.g., wait some time for the weather to change).
Next, we distinguish between the appearance of initial, goal, and mixed cores

Initial and Goal Cores

We know that initial and goal cores describe trivial unsolvable subtasks. These MUSs can be easily transformed to a comprehensive output. In order to completely explain an unsolvable planning task, we have to mention all initial and goal cores as indicated in Example 2.3.1.

Definition 4.1.1 (Output if the empty core was generated)**.** *»All plans have been considered.«*

Definition 4.1.2 (Output if an initial core was generated for plan horizon ph)**.** *»We cannot leave the initial configuration specified by the initial core for ph many steps.«*

Definition 4.1.3 (Output if a goal core was generated for plan horizon ph)**.** *»We cannot reach the goal configuration specified by the goal core in ph many steps.«*

Example 4.1.1 (Output of an initial and goal core)**.** *The last iteration of BUS on the unsolvable pegsol-invasion planning task in Example 3.2.1 returned an initial and goal core. The output of the initial core is:*

»We cannot leave the initial configuration of l_4 for 2 steps.«

The output of the goal core is:

»We cannot reach the goal configuration of l_4 in 2 steps.«

Each witness may be generated in several iterations throughout BUS and has a smallest plan horizon ph_{min} for which it was generated first. We propose to return all witnesses ordered by their size (the number of clauses), starting with the smallest one, and if several of the same size exist, we order these by their corresponding ph_{min}, starting with the smallest one.

Mixed Cores

Up to the point where we discover a witness for plan infeasibility, we obtain only mixed cores. We take a closer look to those mixed cores and divide them further into three classes. This classification is motivated by the fact that some fluents are shared among the initial and goal configurations in subtasks and some are not. A *pair* of fluents indicates that no *ph*-plan exists if we specify that the fluent starts with a certain assignment and ends with another one. Therefore, a *ph*-plan exists leaving the original initial assignment or reaching the original goal assignment of that fluent. A fluent which does not appear as a pair is called a *single*.
Singles which belong to a pair in the original planning task indicate difficulties in this planning task. They state that, because a fluent is assigned to a certain value initially or is requested to contain a certain value at the end, they prohibit other fluents to be fulfilled in *ph* many steps. Even if these difficulties might be resolved by analyzing higher plans, they still signal sources of problems for the overall planning task to the user.
Let us specify a second definition for the set of fluents in the initial and goal configuration:

$$\varphi_{initial} \cup \varphi^{ph}_{goal} := \varphi^{ph}_{singles} \cup \varphi^{ph}_{pairs}$$

Definition 4.1.4 (Classes of mixed cores)**.** *Three classes for a mixed core C in the context of Planning exist.*

- If $\varphi^{ph}_{C,singles}$ is empty, we call C a *pair core*.
- Else if $\varphi^{ph}_{C,pair}$ is empty, we call C a *single core*.
- Else we call C a *double mixed core* (because it is mixed in terms of initials and goals, and singles and pairs).

Definition 4.1.5 (Output if a pair core was generated for plan horizon *ph*)**.** *»The combination of all pairs prohibits a ph-plan for them.«*

Definition 4.1.6 (Output if a single core was generated for plan horizon *ph*)**.** *»The combination of all singles prohibits a ph-plan for them.«*

Definition 4.1.7 (Output if a double mixed core was generated for plan horizon *ph*)**.** *»Because of the singles no ph-plan exists for the combination of pairs.«*

Example 4.1.2 (Output of a pair and double mixed core)**.** *Among the MCSs and MUSs generated in Example 4.0.2 we find pair and double mixed cores. For example, the first iteration returns the double mixed core* {1,2,4} *stating that:*

»Because location l_2 is initially filled no 1*-plan exists for clearing location l_1. But there exists a* 1*-plan clearing location l_1.«*

The second iteration returns the following output for the pair core:

»The combination of clearing locations l_1 and l_2 prohibits a 2*-plan for them.«*

Similar to witnesses, each mixed core has a ph_{min} for which it appears first. We propose to return all classes of mixed cores ordered by their size (the number of

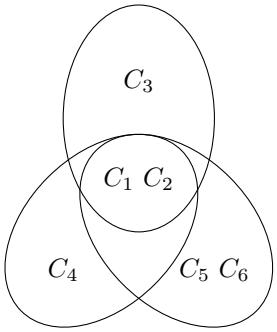


Figure 4.2: The three MUSs are $\{C_1,C_2,C_3\}$,$\{C_1,C_2,C_4\}$,$\{C_1,C_2,C_5,C_6\}$. The only shared set is $\{C_1,C_2\}$ and the three support sets are $\{C_3\}$, $\{C_4\}$ and $\{C_5,C_6\}$.

clauses), starting with the greatest one, and if several of the same size exist, we order these by their corresponding ph_{min}, starting with the smallest one.

Additionally to the plain MUSs we present a source of information combining different MUSs. It is possible to create a *shared set* of clauses which is a non-empty subset of more than one MUS. Once a shared set has been detected we can specify for each relevant MUS the *support set* as the remaining clauses in the MUS with respect to the shared set. All support sets indicate why the shared set is not solvable. This information is more general than the plain MUS, because several MUSs are grouped into one shared set along with all support sets. A visualization of how MUSs, a shared set and all relevant support sets are connected can be found in Figure 4.2.

Definition 4.1.8 (Shared Set and its Support Sets)**.** *A shared set $S \subset M_i$ is a non-empty subset of each $M_i \in \mathcal{M}_{SHA} \subseteq \mathcal{M}$ with $|\mathcal{M}_{SHA}| > 1$ and $\mathcal{M}$ the set containing all generated MUSs in one iteration of BUS. The support sets consist of all sets $M_i \setminus S$ with $M_i \in \mathcal{M}_{SHA}$.*

Definition 4.1.9 (Output if a shared set was detected for plan horizon ph)**.** *»The support sets prohibit a ph-plan for the shared set.«*

Example 4.1.3 (Output of a shared set)**.** *In the third iteration of Example 4.0.2 we detect the shared set* {1,4} *along with the support sets*

$$\{\{2,3\},\{2,5\},\{2,6\},\{3,5\},\{3,6\},\{5,6\}\}$$

generating the following output:

»Each combination of initial and goal specification of locations l_2 and l_3 prohibits a 3*-plan for the subtask freeing l_1.«*

We propose to return all shared sets ordered by their size (the number of clauses), starting with the greatest one, and if several of the same size exist we order these by one of several possible criteria.

1. Order by descending $\operatorname{argmax}_i |\mathrm{SUP}_i|$ (*maximum support set*), the sizes of the greatest support set for each shared set.

2. Order by descending $\Sigma_i|\text{SUP}_i|$ (*accumulative support sets*), the accumulative sizes of all support sets for each shared set.

3. Order by descending |SUPs| (*amount of support sets*), the amount of support sets for each shared set.

4.2 Solvable Subtasks

Complementary to explaining unsolvable subtasks with MUSs it is possible to present solvable subtasks with the MCSs obtained in the first phase of the UNSAT core generation algorithms. We obtain a set of MSSs via the complements of all MCSs. Further we do not consider all MSSs which describe trivial subtasks. These solvable trivial subtasks do only state that we can explore the state space from the initial or reach the goal configuration, which is unrelated to the given planning task.

Definition 4.2.1 (Output of a maximal solvable non-trivial subtask for plan horizon *ph*)**.** *»The specified subtask is solved by a ph-plan.«*

Alternative assignments for the fluents of the original planning task which do not appear in the subtask exist and can be returned by analyzing the satisfying assignment. Note that we have no control about which assignment is chosen by the SMT solver for all free variables. A better alternative is to return an expression describing all possible values for the corresponding fluent. This is trivial for Boolean variables (as we need only to flip the assigned truth value) but difficult for, e.g., variables over reals and requires *quantifier elimination* techniques [Nip10].

Example 4.2.1 (Output of a maximal solvable subtask)**.** *Throughout applying BUS on the unsolvable chessboard-pebbling instance in Example 4.0.2 we collect several maximal solvable subtasks. Examples are subtask* {1,2,4} *(second iteration) with output:*

»The subtask with initially filled locations l_1 and l_2, and clearing location l_1 is solvable with a 2-plan.«

Further, subtask {1,2,3,5,6} *(third iteration) with output:*

»The subtask with initially filled locations l_1,l_2 and l_3, and clearing locations l_2,l_3 is solvable with a 3-plan.«

Throughout BUS, we generate at most as many maximal non-trivial solvable subtasks as we obtain MCSs. Similar to witnesses, each MSS has a ph_{min} for which it appears first. We propose to return all subtasks ordered by their number of initial configurations, starting with the greatest one, and if several of the same size exist we order these first by their greatest overall size, and secondly by their corresponding ph_{min}, starting with the smallest one.
We define this order motivated by the fact that the larger a subtask is, the closer it is to the original planning task, and the shorter the solving plan is, the less unnecessary actions are performed in order to reach the goal configurations.

Example 4.2.2 (Output of the most helpful maximal solvable non-trivial subtask)**.** *Following the described principle, we output the maximal solvable non-trivial subtasks* {1,2,3,5,6}*, due to the full initial configuration, first.*

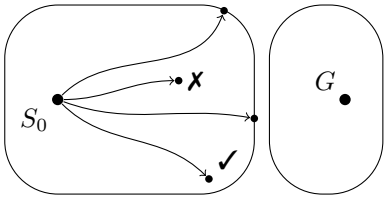


Figure 4.3: Subtasks which are executable in S_0 are marked with ✓. The ones which are not executable are marked with ✗, or reach the border of the set of reachable states from S_0.

4.3 Reasons of Infeasibility

In this section we present a *reason* of infeasibility for some planning task. A reason describes an action, necessary to solve a subtask, which is not executable in some state reachable from the original initial configuration. The motivation to return that specific action is the possibility that this action is *necessary* to solve the original planning task. Note that it cannot be ensured that the detected action is truly necessary. Depending on the actual subtask and plan the detected reason might be not helpful for a user.
Listing all reasons can be seen as an exhaustive search in the state space, where we check for each state S if some actions are executable in S. If no action is executable in S and S is reachable from S_0, then S is on the *border* of reachable states (Figure 4.3).

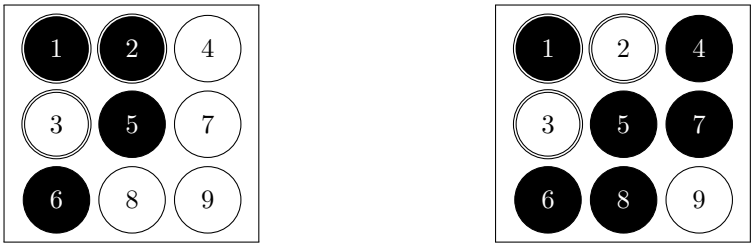
Definition 4.3.1 (Reason R for infeasibility of a planning task)**.** *A reason R is a pair (S,α) with S a world state reachable from S_0 with a ph-plan and $\alpha \in \mathcal{A}$ an action. No assignment θ exists such that α is executable in S. The set of all reasons is denoted as $\mathcal{R}$.*

Example 4.3.1 (Helpful and non-helpful reasons)**.** *First, we present a helpful reason. Imagine that in order to reach a destination we need to pass a bridge which is currently maintained. A subtask of the given planning task can ignore the fact that the bridge is down and requires passing the bridge. If we analyze the plan solving the subtask in the original planning task we detect that it is not possible to pass the bridge.*

Next, we describe a non-helpful reason. Now we consider the possibility that the subtask ignores the initial position. If the corresponding plan solving the subtask consists only of entering the building at the destination, we detect the reason that we cannot enter the building because we are not in front of it.

We generate a (possibly empty) set of reasons $\mathcal{R}_{subtasks} \subseteq \mathcal{R}$ by executing the plans of the maximal solvable non-trivial subtasks from the Section 4.2. It is possible that the *ph*-plan solving a subtask is completely executable in the original planning task. In this case we conclude that there does not exist any reason not to reach the subtask in the context of the original planning task.
The other possibility is that for some $ph^\dagger < ph$ the $ph^{\dagger,th}$-action is not executable in $S_{ph^\dagger-1}$. We detect such an action by the following procedure:

1. Loop over all maximal solvable non-trivial subtasks obtained from the original planning task provided by BUS.



(a) Reason appears. (b) No reason appears.

Figure 4.4: World states of the domain chessboard-pebbling appearing in reasons for infeasibility. In (a) freeing l_1 is not possible after freeing l_2 because of the initially filled location l_3. In contrast, the world state in (b) is the goal state of the subtask.

2. Generate for each subtask all possible plans. This is realized by adding a blocking clause for each found plan and solving the encoding extended by the blocking clauses until it becomes unsatisfiable. The generated plans are added to the set of all plans (each plan only needs to be checked once even if it solves several subtasks).

3. Loop over all obtained plans.

4. Create for the current plan an encoding containing the original initial but no goal configuration along with enforcing the first n actions (starting with $n = 1$) of the plan.

5. If the encoding is satisfiable increase n until it reaches the end of the plan. Otherwise analyze the assignment of the encoding in step $(n - 1)$ and add the detected reason R to $\mathcal{R}_{subtasks}$.

Example 4.3.2 (Reasons in the chessboard-pebbling planning task)**.** *We reconsider the solvable subtasks of Examples 4.2.1,4.2.2. The subtasks specified by the MSS* {1,2,4} *has a reason:*

- *The* 1*-plan is freeing location* l_2*.*
- *World state* S *is defined by a set of fluents which assignments are indicated in Figure 4.4a.*
- *The action* α *which is not executable is freeing location* l_1*.*

The other subtasks specified by the MSSs {1,2,3,5,6} *and* {2,3,4,5,6} *are examples for executable subtasks by executing (for both subtasks) the following actions (other partial orders exist): free* l_2*, free* l_5*, and free* l_3 *resulting in the world state indicated in Figure 4.4b. Because these two subtasks are executable in the original planning task, we can return them to the user.*

Chapter 5

Evaluation

This chapter presents evaluations of the approaches and algorithms presented throughout this thesis. First, we analyze the MUS generation algorithms *CAMUS* and *BLS* with respect to two different SMT solvers *Z3* and *SMT-RAT* in Section 5.1 in order to determine the most suitable SMT solver and MUS generation algorithm for the later experiments. The benchmarks consist of general UNSAT SMT-LIB encodings, as well as hand-crafted encodings of well-known planning domains.
Next in Section 5.2, the planning domains used in the experiment before are solved by the approach BUS, and compared with the original BMC approach and state-of-the-art planners focusing on proving plan infeasibility. The aim is to conclude if BUS can compete with those tools which are not interested in explaining the unsolvable planning tasks. Moreover, we analyze the relation between the solving time and the amounts of generated MCSs and MUSs.
Finally, we run the tool `PEX` (*Planning EXplainer*) on the same planning domains in Section 5.3 and give examples how we can explain unsolvable planning tasks by returning a witness of infeasibility (if found before the timeout), mixed cores, a shared subset, a maximal non-trivial solvable subtask for which no reason exists and a reason.

5.1 UNSAT Benchmarks

For this experiment we use benchmarks from the web-based service *StarExec* [SST14] and consider encodings representing domains of the *Unsolvability International Planning Competition Track* (IPC)[3].

5.1.1 Encodings

The platform StarExec[7] is used to test and compare tools for various techniques, not only SAT and SMT, but also *theorem provers*, *constraint solvers*, *model checkers*, and *software verifiers*.
We will use the *QF_LRA*, *QF_LIA*, and *QF_LIRA* benchmarks from the *Unsat-Core Track*. Note that we only use a subset of *QF_LIA* benchmarks due to the huge number of instances. In the *IPC* several planners competed in proving infeasibility on instances of well know planning domains. We use four of these domains and generate for each domain encodings depending on various plan horizons. In total we run the tools on 3388 benchmarks, 2453 from StarExec and 935 from IPC.

	$score_{Z3}$	$score_{SMT-RAT}$	draws	timeouts
all	2474	620	15	279
StarExec	1544	615	15	279
IPC	930	5	0	0

Table 5.1: Scores of SMT solvers SMT-RAT and Z3

	$score_{\texttt{CAMUS}}$	$score_{\texttt{BLS}}$	draws	timeouts
all	1369	1135	7	877
StarExec	581	1118	7	747
IPC	788	17	0	130

Table 5.2: Scores of MUS generation algorithms `CAMUS` and `BLS`

We are interested in comparing the solvers and the MUS generation algorithms. For all experiments we compare two approaches and plot a graph where the x-axis is used for one tool's solving times, and the y-axis for the second tool's solving times. The diagonal line indicates instances where draws (both tools solved with the same speed) appeared. All marks in the upper triangle state that the approach of the x-axis was faster and all below state the same for the approach of the y-axis. Note that even if we used a one-minute timeout the system does not state that a timeout appeared but prints a time close to 60 seconds (for example 59.5s). Thus, we treat all solving times above $59s$ as timeouts.

5.1.2 Results

We evaluate the solvers SMT-RAT and Z3. The results are visualized in Figure 5.1 and the scores are displayed in Table 5.1. The SMT solver Z3 solves about 79% of all benchmarks faster than SMT-RAT (disregarding timeouts). If we consider only the encodings representing planning instances from the IPC the value increases even to 99%. This makes Z3 a convenient solver for the next experiments.

Next, we compare the MUS generation algorithms CAMUS and BLS based on Z3 on all benchmarks. The results are visualized in Figure 5.2 and the scores are displayed in Table 5.2. On the complete set of benchmarks, the tool CAMUS performs better on about 54% (disregarding timeouts). This does not indicate that one of the two algorithms is truly better than the other. But if we distinguish the benchmarks, we see that BLS generates cores faster on 65% of the StarExec benchmarks and CAMUS on 97% of the IPC benchmarks. Therefore, we use the combination of Z3 and CAMUS for BUS in the next experiment.

5.2 Deciding Planning Tasks

We use 16 unsolvable and 4 solvable planning task instances from four domains of the IPC benchmarks from ICAPS 2016[3] to evaluate BUS. A git repository is available containing all planners and domains used during the IPC[10].

[10] https://bitbucket.org/planning-researchers/unsolve-ipc-2016

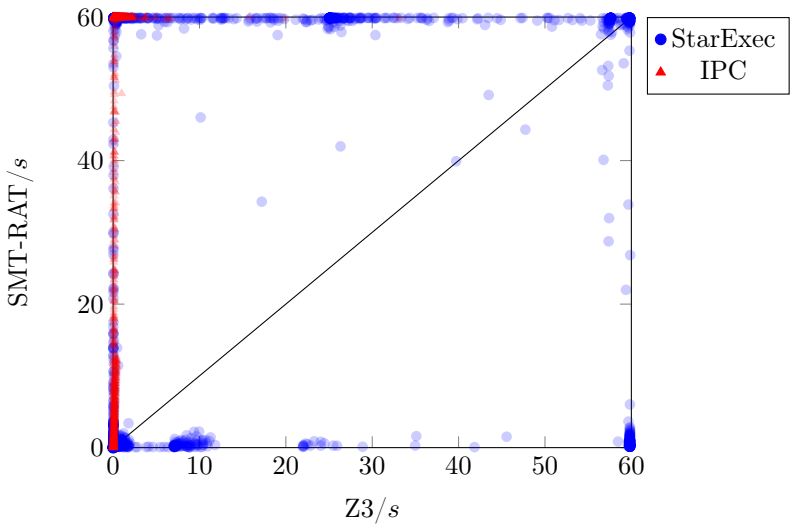


Figure 5.1: Comparison of SMT solvers SMT-RAT and Z3

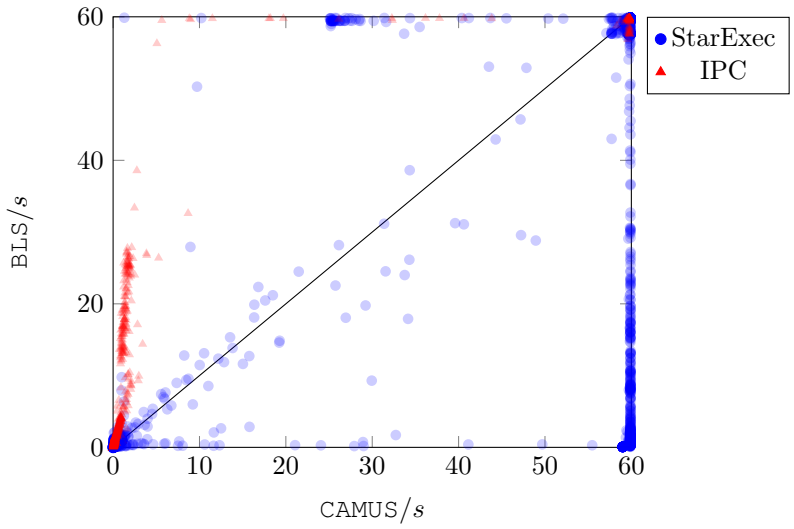


Figure 5.2: Comparison of MUS generation algorithms CAMUS and BLS

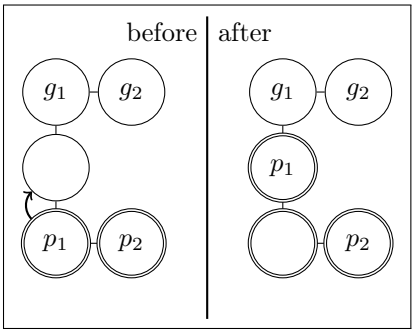


Figure 5.3: bottleneck01-cus

5.2.1 Domains

Four domains are encoded manually into SMT-LIB files. Among them are the domains *bottleneck*, *chessboard-pebbling*, *pegsol-invasion* and *slidingtiles*. For some domains we added custom instances of reduced complexity. In addition to unsolvable instances we test solvable instances as well.
We encapsulate each planning task in a `Python` script which we call a *generator*. For a given plan horizon *ph* the generator creates an SMT-LIB encoding representing the problem instances such that satisfiability means that a plan of length *ph* exists. If the encoding is unsatisfiable no *ph*-plan exists. These generators are required as BUS increases the plan horizon in each step.
We do not make use of available translators as they have several problems when tasked with our use-case. Some return bloated encodings because they offer functionalities we are not interested in (for example, *SMTPlan* [CFLM16] enables planning in hybrid systems). Therefore, using those encodings would increase the solving time unnecessarily. A second problem is the impossibility of marking clauses as clauses of interest before the translation. It is possible to post-process the encodings, but this requires analyzing the automatically created encodings. Finally, the translator needs to be able to extend the basic encoding with the information to omit loops, to ensure the properties BUS demands for as described in Section 3.

Bottleneck

The bottleneck domain consists of a set of locations which are connected and a number of persons which try to reach goal locations. No location can be visited twice and the available connections between locations can prohibit a solution. A visualization of an action in the domain can be found in Figure 5.3. This domain has no need for additional clauses because the number of visited states grows with each step in a plan. The bottleneck planning task can be found in the directory `domains/FINAL/bottleneck/` of the git repository[10]. Instances differ in the size of persons, locations, and connections between locations and are described in the Appendix A.1.

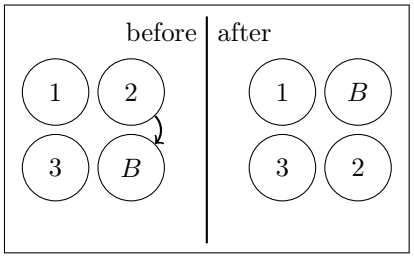


Figure 5.4: slidingtiles1

Pegsol-Invasion

The pegsol-invasion domain was already introduced in Example 2.2.1. It consists of filling a set of locations by making use of the following rule: If two locations next to each other are filled then a third free neighboring location forming a line with the first two ones can be filled and the two filled locations are cleared. This domain has no need for additional clauses because the number of filled locations decreases with each step in a plan. The most popular field for pegsol is a cross of width 3. The pegsol-invasion version uses a rectangle with the top filled with free locations. The pegsol-invasion planning task can be found in the directory `domains/FINAL/pegsol-row5/` of the git repository[10]. Instances differ in the size of the rectangle, initially filled and desired free locations and are described in the Appendix A.2.

Chessboard-Pebbling

The chessboard-pebbling domain was already introduced in Example 4.0.1. It consists of clearing a set of locations by making use of the following rule: A filled (not free) location can be cleared by turning the free locations below and right into filled ones. This domain has no need for additional clauses because the number of filled locations grows with each step in a plan. The chessboard-pebbling planning task can be found in the directory `domains/FINAL/chessboard-pebbling/` of the git repository[10]. Instances differ in the size of locations, initially filled and desired free locations and are described in the Appendix A.3.

Slidingtiles

The slidingtiles domain consists of reaching a desired placement of tiles by moving a tile to the neighboring (left, above, right, and below) blank location. This domain needs the additional clauses prohibiting loops as the actions allow to move the same tile back and forth, thus entering a loop. The most popular field for slidingtiles is a 3×3 grid with 8 tiles (indicated by numbers from 1 to 8). A visualization of an action in the domain can be found in Figure 5.4. The slidingtiles planning task can be found in the directory `domains/FINAL/slidingtiles/` of the git repository[10]. Instances differ in the size of the rectangle, the initial and goal mapping of the tiles and are described in the Appendix A.4.

5.2.2 Results

The detailed results are displayed in Appendix B.1. We analyze three properties of BUS. First, the additional computational cost of generating all MUSs. Next, we compare the state-of-the-art planner SymPA with BUS in terms of solving speed. Finally, we investigate the solving speed, and amount of MCSs/MUSs progression over the iterations of BUS.

Additional Computational Cost of generating all MUSs

To begin with, we analyze the additional computational cost caused by generating all MUSs of an unsatisfiable encoding. We test as many encodings as possible within a one-minute timeout, with generating all MUSs and without. Because we can test this procedure only on unsolvable instances, we omit the four solvable instances in this experiment. The two variants represent the efforts BUS (generating all MUSs) has to make compared to the original BMC planning approach (not generating all MUSs). The amounts of checked plan horizons for both variants are visualized in Figure 5.5 and displayed in Table B.1.
As expected, the second variant reaches a higher plan horizon as the first one. Note that this does not directly state that the original BMC planning approach outperforms BUS, because it is possible that the necessary completeness threshold is still higher than the reached plan horizon, but BUS reaches already a lower plan horizon stating plan infeasibility. This is the case for the instance chessboard-pebbling1. BUS states plan infeasibility within the one-minute timeout. And the second variant reaches iteration 332, but needs to reach iteration 512 to state plan infeasibility as described in Example 4.0.2.
The plot shows a linear dependency between the two variants which is closer to the diagonal line than one might expect. This is due to the complexity of the planning tasks. As we can see in Table B.1, infeasibility has been detected early for all instances which have been solved within the timeout. Apart from the instance pegsol-invasion2' the witness was close to detecting the empty core (or it was the empty core) which reappears from this point on. This explains the observed dependency. In order to detect the empty core, CAMUS only requires one solver call.
As stated in Section 2.3.2, we decided to introduce clauses of interest from a computational point of view. We see already that we check less iterations when marking all initial and goal specifications instead of a subset of those. Moreover, if we use the original CAMUS algorithm (all clauses are regarded as clauses of interest and used for generating all MUSs) we are not able to check even one iteration for all instances. Therefore, the assumption proposed in Section 2.3.2 is justified.

Comparison of BUS and SymPA

Next, we compare the solving times of the state-of-the-art planner SymPA [Tor16] and BUS. The comparison of solving times is visualized in Figure 5.6 and displayed in Table B.2. It is no surprise that SymPA outperforms BUS. Moreover, we see that BUS is not able to detect plan infeasibility of planning tasks where SymPA has no problems.
This result highlights the possibility to use both procedures in parallel. Thus, even if we cannot state plan infeasibility and only collect information in terms of XAI with BUS, we can still answer the question whether the planning task is unsolvable.

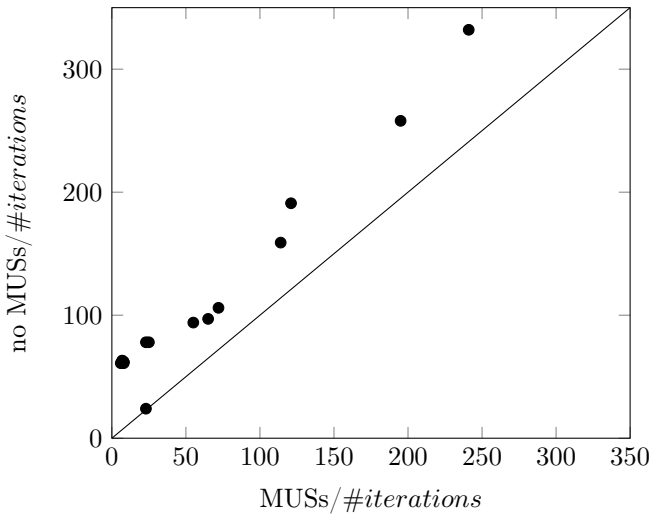


Figure 5.5: Amounts of iteration reached within one-minute, with generating all MUSs and without.

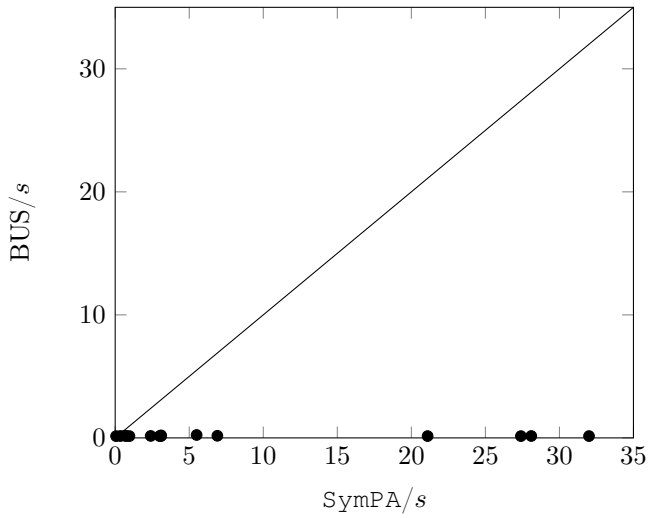


Figure 5.6: Comparison of BUS and SymPA on deciding planning tasks.

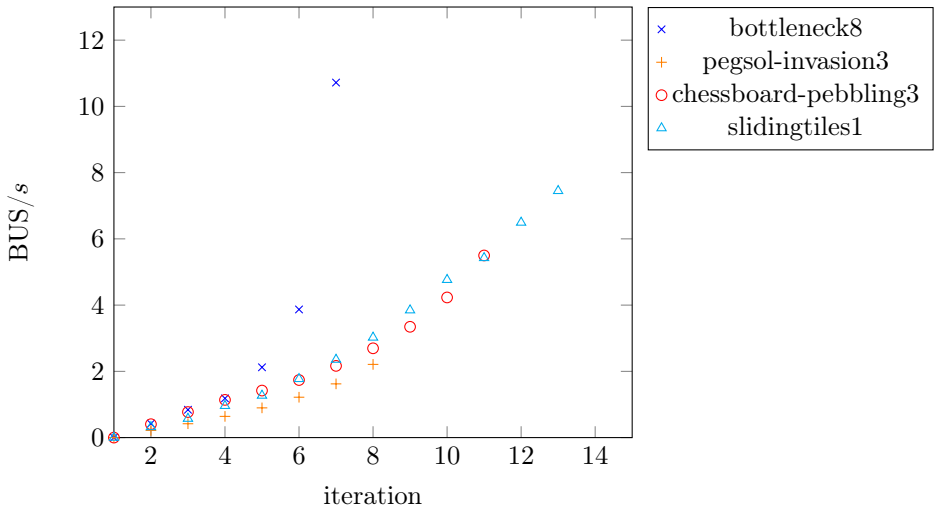


Figure 5.7: BUS solving time progression depending on the iterations.

Progression of Solving Time and generated MCSs, MUSs of BUS

We display the amount of generated MCSs and MUSs of BUS, along with the number of clauses of interest and actions in Tables B.3, B.4. Additionally, we plot the progression of the solving time, the amount of MCSs, and the amount of MUSs depending on the iterations in Figures 5.7,5.8, 5.9. For this experiment we use one instance of each domain: bottleneck8, pegsol-invasion3, chessboard-pebbling3, slidingtiles1. The dependency of all generated MCSs and MUSs with the solving time is plotted in Figures 5.10, 5.11.

The progression of properties dependent on the iterations indicate a linear dependency. Only for the solving time (especially for the instance bottleneck8) an exponential dependency due to the increasing encoding seems possible. In the case of MCSs and MUSs we observe that the amount of discovered MCSs and MUSs stays constant throughout BUS.
This indicates that some MCSs and MUSs are detected multiple times (e.g., for the slidingtiles1 instance where the MCSs and MUSs of complete iterations appear again). It needs to be investigated if there are some principles on which the MCSs and MUSs in the context of Planning are based on. If such exist, it might be possible to precompute a set of MCSs and MUSs, accelerating the overall procedure.
The amount of all generated MCSs and MUSs (Figures 5.10, 5.11) indicates a linear dependency on the solving time of BUS for each instance. This underlines the result of the MCSs and MUSs progressions over the iterations. Not only for each iteration we obtain a constant amount of MCSs and MUSs, but also for different instances of the same domain the difference is constant.
One explanation for the constant amount of new MCSs and MUSs is that the number of clauses of interest stays constant as well. Thus, the solving time increases because the encoding becomes bigger, but not the amount of MCSs and MUSs.

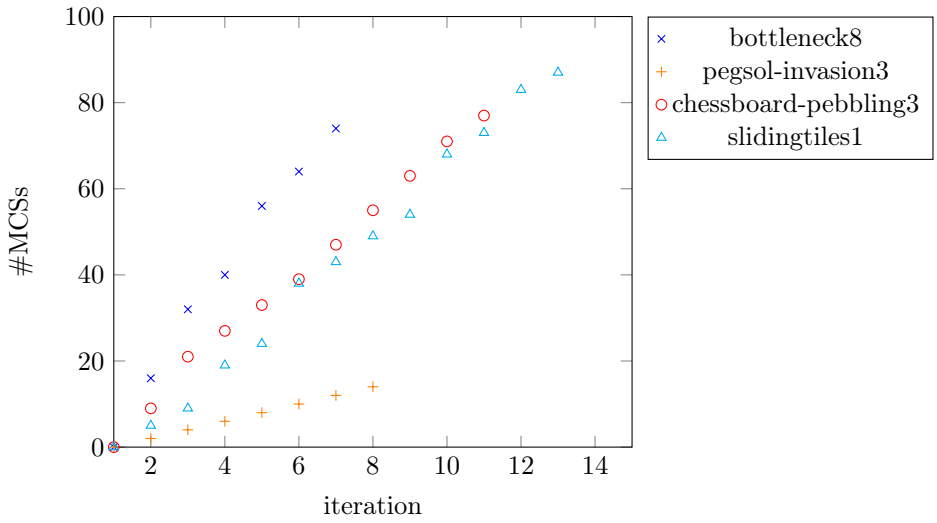


Figure 5.8: Progression of generated MCSs depending on the iterations.

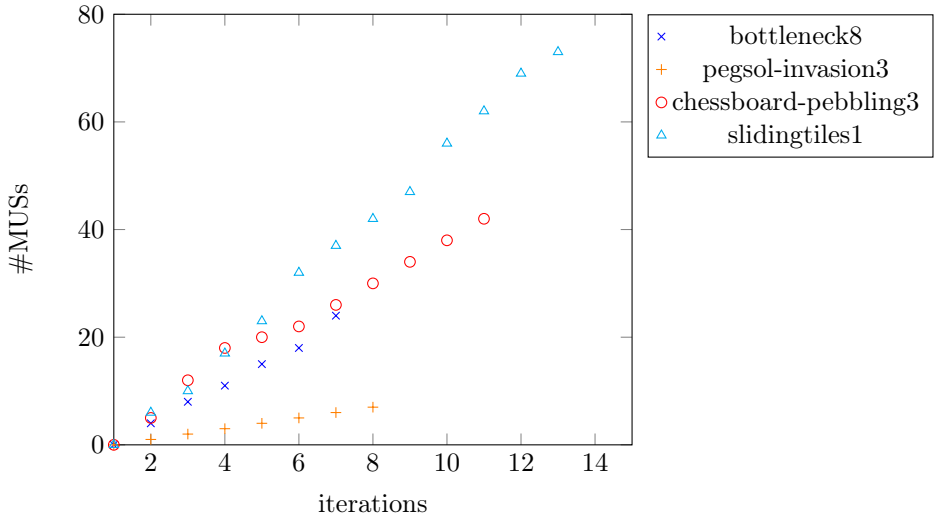


Figure 5.9: Progression of generated MUSs depending on the iterations.

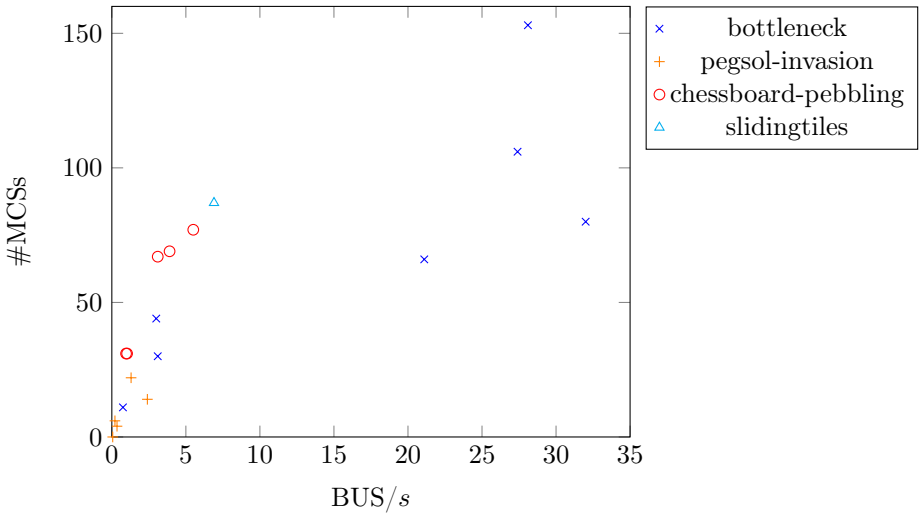


Figure 5.10: Amount of MCSs dependent on BUS solving time

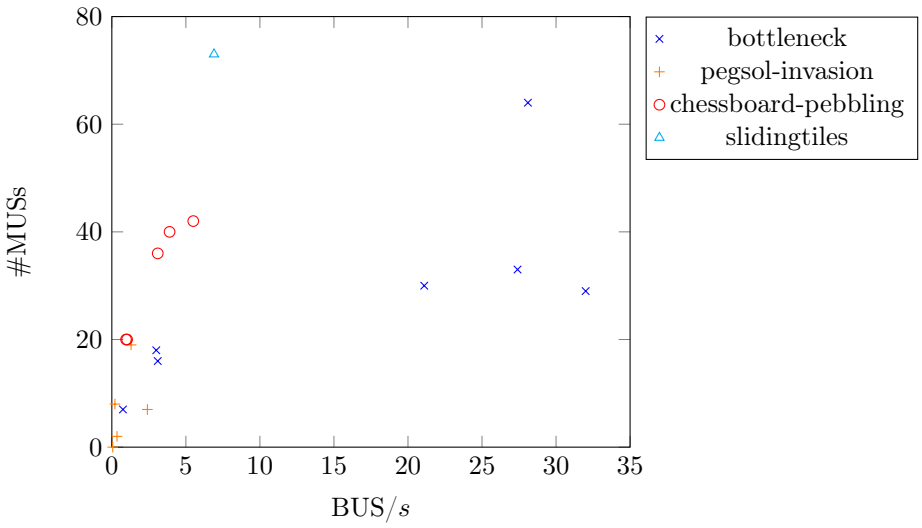


Figure 5.11: Amount of MUSs dependent on BUS solving time

5.3 Explaining Planning Tasks

In this section we present outputs of the tool PEX encapsulating BUS and evaluating the generated MCSs and MUSs. We analyze the output of PEX on four instances of the same domains used in the second experiment in Section 5.2.

5.3.1 Results

The output of PEX consist of the following components:

1. Witnesses (if found before the timeout)
2. Pair, single and double mixed clauses
3. Shared sets along with their support sets
4. Maximal solvable non-trivial subtasks with executable plans in S_0
5. Reasons

We present this information for one instance of each domain: bottleneck3, pegsol-invasion1, chessboard-pebbling1, slidingtiles1. Some outputs of the chessboard-pebbling domain have been analyzed already in Chapter 4 and is reused in this evaluation besides adding new output.

bottleneck3

For instance bottleneck3 we obtain:

- Witnesses ({2}):

  »We cannot leave the initial configuration where p_2 starts in location 2 for 6 steps.«

- Pair core ({1,2,4,5}):

  »The combination persons p_1,p_2 moving from location 1 respectively 2 to 7 respectively 8 prohibits a 4-plan for them.«

- No single core:
- Double mixed core ({2,3,6}):

  »Because of person p_2 starting in location 2 no 4-plan exists for person p_3 moving from 3 to 9.«

- Shared sets: We obtained two shared sets of size two {1,4} and {3,6} with both having the same value for criteria maximum support set and accumulative support sets. Only criterion amount of support sets distinguishes them, returning {3,6} with support sets {1,4},{2},{5} first.

  »Person p_1 moving from 1 to 7, person p_2 starting in 2, and moving to 8 prohibits a 4-plan for person p_3 moving from 3 to 9.«

The other shared sets {1,4} with support sets {2,5},{3,6} yields the following output:

»Person p_2 moving from 2 to 8, and person p_3 moving from 3 to 9 prohibits a 4-plan for person p_1 moving from 1 to 7.«

- Maximal solvable subtask ({1,2,3,4}):

  »Letting reach person p_1 its goal and using the original positions of the other persons without specifying any goal for them is solved by a 2-plan.«

- Reason ({1,3,4,5}):

  »It is not possible for p_2 to move left in the initial configuration.«

pegsol-invasion1

For instance pegsol-invasion1 we obtain:

- Witnesses ({1,3,4}):

  »We cannot leave the initial configuration where l_1,l_3,l_4 are free for 1 step.«

- No pair core:
- Single core ({3,4,7}):

  »The combination initially free locations l_3,l_4 and filling l_1 prohibits a 1-plan for them.«

- Double mixed core ({1,2,7}):

  »Because of location l_2 is initially free no 1-plan exists filling l_1.«

- Shared set {3,7} with support sets {1},{4},{5}:

  »The initial configuration of locations l_1,l_4,l_5 prohibits a 1-plan for obtaining a filled location l_1 with location l_3 initially free.«

- Maximal solvable subtask (1,4,5,6,7) (Not executable in S_0):

  »Using the original planning task without specifying the initial configuration of locations l_2,l_3 is solved by a 1-plan.«

- Reasons ({1,3,6,7}):

  »It is not possible to jump from l_5 over l_4 to l_3 in the initial configuration.«

chessboard-pebbling1

For instance chessboard1 we obtain:

- Witnesses (empty core):

  »All plans have been considered.«

- Pair core ({2,3,5,6}):

  »The combination of freeing locations l_2,l_3 prohibits a 1-plan for them.«

- No single core:
- Double mixed core ({1,2,3,4}):

  »Because of the initially filled locations l_1,l_2,l_3 no 2-plan exists for freeing l_1.«

- Shared set {1,4} with support sets {{2,3},{2,5},{2,6},{3,5},{3,6},{5,6}}:

  »Each combination of initial and goal specification of locations l_2,l_3 prohibits a 2-plan for the subtask freeing l_1.«

- Maximal solvable subtasks ({1,2,3,5,6}):

  »The subtask with initially filled locations l_1,l_2 and l_3, and clearing locations l_2,l_3 is solvable with a 3-plan.«

- Reasons ({1,3,4,5}):

  »We cannot free location l_1 after freeing location l_3 because l_2 is still filled.«

slidingtiles1

For instance slidingtiles1 we obtain:

- Witnesses (empty core):

  »All plans have been considered.«

- Pair core ({2,4,5,7}):

  »The combination of changing location top-right from 2 to 1, and location bottom-right from B to 3 prohibits a 2-plan for them.«

- Single core ({1,3,5,7}):

  »The combination of the initial locations top-left, bottom-left, and obtaining tile 1 at location top-right and tile 3 at location bottom-right prohibits a 1-plan for them.«

- Double mixed core ({2,3,4,6,7}):

»Because starting with 2 at the top-right no 5-plan exists for changing the bottom tiles from 3 to 2 and B to 3.«

- Shared set {6,7} with support sets {1,2},{1,3},{1,4},{2,3,4}:

  »The initial configuration combinations of the top-line, left-line, top-left-diagonal, and all locations except the top-left prohibits a 5-plan for obtaining 2 at the bottom-left and 3 at the bottom-right.«

- Maximal solvable subtasks {1,2,3,4,5,6}:

  »Using the original planning task without specifying the goal configuration of the bottom-right location (3) is solved by a 4-plan.«

- Reasons ({2,3,5,6,7}):

  »It is not possible to move the bottom-left tile to the top in the initial configuration.«

As stated before, it is difficult to define at what point the returned output is helpful for a user or not. Every user might have different opinions about how helpful some information is, depending on its current situation. Therefore, the only possibility to state that generated information for explaining unsolvable planning tasks is helpful or not in terms of trust, interaction and transparency, is to make a survey with humans. Properties of `PEX` which can be investigated are:

1. Is the explanation helpful or not?
2. Does the user know this already, or is the information new?
3. Is the explanation presented in an understandable way, or does the user have to use additional efforts?
4. Are there too many explanations or not enough?

The first property is crucial in terms of XAI, as a system which outputs only information which is not helpful, or trivial, does not explain the detected problem to users. Improvement can only be achieved by defining new heuristics deciding which MCSs and MUSs to use for explanations and how to interpret them. The second property is more difficult to tackle, as it is challenging for a system to predict the user's knowledge. This point can be improved by asking the user to input his knowledge about the planning task before the evaluation. An understandable explanation is related to the topic of Plan Explanation [SBM11], [CFT+18]. If we output too much information we need again to use heuristics in order to decide what information to use.

All returned reasons (except of the chessboard1 instance) only indicate actions which cannot be executed in the initial configuration. This information is obvious and does not explain the planning task as it was intended. Besides the difficulty to output helpful reasons, the other information highlights valuable information about the planning task. For example, the shared set for the pegsol-invasion instance indicate which specific locations cause problems (and which do not). A second example is the maximal solvable subtask for the slidingtiles instance. The fact that not pursuing a goal location for one specific tile yields a solvable planning task, is not directly accessible to human users.

Chapter 6

Conclusion

6.1 Summary

In this Master Thesis we tackled the question why some planning task is unsolvable in terms of XAI. We applied tools of SMT, namely generating all MUSs, and the related Planning as Satisfiability and BMC paradigm, in order to investigate which parts of the planning task cause the problem and which parts are still solvable.
During our research we discovered, MUSs in the context of Planning, besides offering valuable information for XAI, enable to detect, whether a planning task is unsolvable or not. The gathered information is similar to the reachability information maintained in other approaches as in [Tor16], [SK16], but collected in a different way, inspired by [Str04]. We proposed an algorithm based on BMC using MUSs, in order to state infeasibility of a planning task, called BUS. The approach of analyzing the MUSs in terms of XAI is implemented in a tool called `PEX` built upon BUS.
The evaluations showed that computational efforts have to be made for generating all MUSs, but at the same time we obtain the possibility to state plan infeasibility in each BMC iteration. However, BUS is clearly outperformed by state-of-the-art planner `SymPA` [Tor16]. Nevertheless, the generated MUSs offer to return a collection of information about the investigated planning task. Note that most of this information is available, even if we cannot state plan infeasibility yet due to computational resources.

6.2 Discussion

The first question arising when investigating problems in XAI is, whether the obtained information is truly *helpful* for a user, interacting with the system. In general, the information about unsolvable subtasks, solvable subtasks and reasons of infeasibility is helpful, because each type of information addresses different facets of the planning task a human user might try to identify as well.
Unsolvable subtasks state, which parts of the planning task are not possible (or not possible yet due to a too small plan horizon), by arguing over parts of the initial or goal configuration. Similarly, the solvable subtasks signal which parts are already solvable. Reasons describe problems of the planning task on the action level. While enrolling the transition scheme of an unsolvable planning task the problem occurs

somewhere between the initial and goal configuration. This picture is similar to the one presented in [Str04].
We reach our target in terms of XAIP, if a user thinks *»I did not know this before and have now better insights about why the planning task is not solvable.«* However, the worst case is not as bad as one might expect. Every chunk of information we return, is based on sound deductions and we do not return wrong conclusions. The less helpful information we might return is *trivial* information, which the user already knows.

6.3 Future Work

After deriving initial results how to use UNSAT cores in order to explain unsolvable planning tasks, there is a lot of potential for future work.
First, the overall algorithm needs to be improved in terms of speed. Even if the main bottleneck is to generate all MUSs, other parts can be accelerated by faster approaches or better maintenance of available information. For example, we use a naive approach of checking whether the plans of solvable subtasks are executable in the original planning task by rechecking each sequence of actions. This can be improved by keeping track of all executable actions. If a sequence of actions has been tested already, we skip these solver calls. Other ways to accelerate the approach are lying in the SMT solver, the hitting set generation or the used data structures. Not only SMT solvers can be used which are faster in general, but also faster for planning tasks in particular. In [Str04] techniques are mentioned to adapt dedicated methods of variable ordering and remembering clauses for the next iteration in the BMC paradigm. A different way to accelerate the process is to use several techniques in parallel [Str04]. One thread could run state-of-the-art planners dedicated to detect plan infeasibility, and `PEX` is run besides to gather information for XAIP.
Next, we need to classify the generated MUSs in terms of information for XAIP. In this work we proposed unsolvable subtasks (which can have the form of witnesses of plan infeasibility or shared subsets), solvable subtasks (which are executable in the original planning task or a reason exists, why this is not the case), and reasons (an action is not applicable in a reachable state). As seen in the evaluations, the displayed reasons are mostly indicating actions, which are not possible in the initial configuration. Here the available reasons have to be analyzed further in order to return more helpful ones. An alternative approach is to consider more clauses as clauses of interest for additional sources of information. However, this depends whether it is possible to decrease the overall computational time of `PEX`.
Finally, the most difficult task in terms of XAIP, is which information to pick for the user, or in which order we display the results, if we decide to provide a set of information. At this point we face the problem, mentioned at the beginning of the thesis, the perception of computer systems and humans differs.

Bibliography

[AT15] Vidal Alcazar and Alvaro Torralba. A Reminder about the Importance of Computing and Exploiting Invariants in Planning. *International Conference on Automated Planning and Scheduling*, 2015:2–6, 2015.

[BBT15] Tomas Balyo, Roman Barták, and Otakar Trunda. Reinforced Encoding for Planning as SAT. *cta Polytechnica CTU Proceedings*, 2:1, 2015.

[Bie09] Armin Biere. Bounded Model Checking. *Handbook of Satisfiability*, 185(99):457–481, 2009.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Frontiers in Artificial Intelligence and Applications*, 185(1):825–885, 2009.

[CFLM16] Michael Cashmore, Maria Fox, Derek Long, and Daniele Magazzeni. A Compilation of the Full PDDL + Language into SMT. *International Conference on Automated Planning and Scheduling*, (Icaps):79–87, 2016.

[CFT+18] Tathagata Chakraborti, Kshitij P Fadnis, Kartik Talamadupula, Mishal Dholakia, Biplav Srivastava, Jeffrey O Kephart, and Rachel K E Bellamy. Visualizations for an Explainable Planning Agent. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 5820–5822. International Joint Conferences on Artificial Intelligence Organization, 2018.

[CGS11] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *Journal of Artificial Intelligence Research*, 40:701–718, 2011.

[Chi08] John W. Chinneck. *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Abraham. SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving. In *Theory and Applications of Satisfiability Testing*, pages 360–368. Springer International Publishing, 2015.

[Coo71] Stephen A Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[CSK17] Tathagata Chakraborti, Sarath Sreedharan, and Subbarao Kambhampati. Balancing Explicability and Explanation in Human-Aware Planning. *CoRR*, abs/1708.0, 2017.

[DB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT Solver. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4963 LNCS:337–340, 2008.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[End01] Herbert B. Enderton. Chapter TWO - First-Order Logic. In *A Mathematical Introduction to Logic (Second Edition)*, pages 67 – 181. 2001.

[ENS92] Kutluhan Erol, D. Nau, and Vs Subrahmanian. When is planning decidable. In *Proc. First Internat. Conf. AI Planning Systems*, pages 222–227. 1992.

[ES03] Niklas Eén and Niklas Sörensson. An EXtensible SAT-solver. *Theory and Applications of Satisfiability Testing*, 2919:502–518, 2003.

[FGFR18] Guillem Franc, Hector Geffner, U Pompeu Fabra, and Miquel Ram. Best-First Width Search in the IPC 2018 : Complete , Simulated , and Polynomial Variants. *International Planning Competition (IPC 2018), Deterministic Part*, 9:23–27, 2018.

[FGS18] Maximilian Fickert, Daniel Gnad, and Patrick Speicher. SaarPlan : Combining Saarland's Greatest Planning Techniques. *Proceedings of the 9th International Planning Competition (IPC 2018)*, pages 10–15, 2018.

[FLM17] Maria Fox, Derek Long, and Daniele Magazzeni. Explainable Planning. *First IJCAI Workshop on Explainable AI (XAI)*, 2017.

[FN71] Richard E Fikes and Nils J Nhsson. STRIPS : A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208, 1971.

[HCZ10] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. A Novel Transition Based Encoding Scheme for Planning as Satisfiability. *Artificial Intelligence*, pages 89–94, 2010.

[Hel06] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[KBC+98] Craig Knoblock, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David E Smith, Ying Sun, and Daniel Weld. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[KJK+12] Geert Kruijff, Miroslav Jan, Shanker Keshavdas, Benoit Larochelle, Hendrik Zender, Nanja Smets, Tina Mioch, Mark Neerincx, Jurriaan Van, Francis Colas, Geert Kruijff, Miroslav Jan, Shanker Keshavdas, Benoit Larochelle, and Hendrik Zender. Experience in System Design for Human-Robot Teaming in Urban Search & Rescue. *Springer Tracts in Advanced Robotics*, 92, 2012.

[KMS96] Henry Kautz, D McAllester, and Bart Selman. Encoding plans in propositional logic. *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR-96)*, pages 374–384, 1996.

[KS17] Joseph Kim and Julie A Shah. Towards Intelligent Decision Support in Human Team Planning. In *AAAI Fall Symposium Series*, pages 5769–5770, 2017.

[KSAH92] Henry Kautz, Bart Selman, Mountain Avenue, and Murray Hill. Planning as Satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363. John Wiley \& Sons, Inc., 1992.

[KT17] Michael Katz and Alexander Tuisov. Adapting Novelty to Classical Planning as Heuristic Search. In *ICAPS*, number Icaps, pages 172–180. 2017.

[LS08] M H Liffiton and K A Sakallah. Algorithms for Computing Minimal Unsatisfiable Sets of Constraints. *Journal of Automated Reasoning*, 40(1):1–42, 2008.

[MSHJ+13] Joao Marques-Silva, Federico Heras, Mikolas Janota, Alessandro Previti, and Anton Belov. On computing Minimal Correction Subsets. *IJCAI International Joint Conference on Artificial Intelligence*, pages 615–622, 2013.

[Nip10] Tobias Nipkow. Linear Quantifier Elimination. *Journal of Automated Reasoning*, 45(2):189–212, 2010.

[NRF+15] Tim Niemueller, Sebastian Reuter, Alexander Ferrein, Sabina Jeschke, and Gerhard Lakemeyer. Evaluation of the robocup logistics league and derived criteria for future competitions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9513:31–43, 2015.

[PM17] Alessandro Previti and Carlos Menc. Improving MCS Enumeration via Caching. In *Theory and Applications of Satisfiability Testing*, pages 184–194. Springer International Publishing, 2017.

[Qui86] J R Quinlan. Induction of Decision Trees. *Machine Learning*, 1:81–106, 1986.

[Rin12] Jussi Rintanen. Planning as Satisfiability : Heuristics. *Artificial Intelligence*, 193:45–86, 2012.

[RW14] Silvia Richter and Matthias Westphal. The LAMA Planner : Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2014.

[SBM11] Shirin Sohrabi, Ja Baier, and Sa McIlraith. Preferred Explanations: Theory and Generation via Planning. *Aaai*, 3:261–267, 2011.

[Sei18] Jendrik Seipp. Fast Downward Remix. *Ninth International Planning Competition (IPC 2018)*, pages 67–69, 2018.

[SH16] Marcel Steinmetz and Jörg Hoffmann. CLone : A Critical-Path Driven Clause Learner. *Proceedings of the 1st Unsolvability International Planning Competition (IPC 2016)*, 2016.

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR*, abs/1712.0:1–19, 2017.

[SK16] Martin Suda and Konstantin Korovin. iProverPlan: a system description. *UIPC 2016 Planner Abstracts*, pages 6–7, 2016.

[SPS+16] Jendrik Seipp, Florian Pommerening, Silvan Sievers, Martin Wehrle, and Chris Fawcett. Fast Downward Aidos. *1st Unsolvability International Planning Competition (IPC 2016)*, pages 28–38, 2016.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicarp, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, (550(7676)):354., 2017.

[SST14] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8562 LNAI:367–373, 2014.

[Str04] Ofer Strichman. Accelerating Bounded Model Checking of Safety Properties. *Formal Methods in System Design*, 24:5–24, 2004.

[Sud14] Martin Suda. Property Directed Reachability for automated planning. *Journal of Artificial Intelligence Research*, 50:265–319, 2014.

[SW97] Klaus Schneider and Holger Weindel. *An Efficient Decision Procedure for S1S*. M. Pfaff., Linz, 1997.

[Tac18] Armando Tacchella. SMarTplan: a Task Planner for Smart Factories Âť. *CoRR*, abs/1806.0:1–16, 2018.

[TLB16] Álvaro Torralba, Carlos L. López, and Daniel Borrajo. Abstraction Heuristics for Symbolic Bidirectional Search. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 3272–3278. 2016.

[Tor16] Alvaro Torralba. SymPA : Symbolic Perimeter Abstractions for Proving Unsolvability. *UIPC 2016 Planner Abstracts*, pages 8–11, 2016.

[Tse83] G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Automation of Reasoning*, pages 466–483. 1983.

[ZSK+17] Yu Zhang, Sarath Sreedharan, Anagha Kulkarni, Tathagata Chakraborti, Hankz Hankui Zhuo, and Subbarao Kambhampati. Plan Explicability and Predictability for Robot Task Planning. In *ICRA 2017 - IEEE International Conference on Robotics and Automation*, pages 1313–1320. Institute of Electrical and Electronics Engineers Inc., 2017.

Appendix A

Instances of Planning Tasks

We describe all instances used throughout the experiments in Section 5.2, 5.3 and provide visualizations.

A.1 Bottleneck

We use the following instances in the bottleneck domain:

1. Instance with two persons and 5 locations (Figure A.1a).
2. Instance with three persons and 7 locations (Figure A.1b).
3. Instance with three persons and 8 locations (Figure A.1c).
4. Instance with four persons and 9 locations (Figure A.1d).
5. Instance with four persons and 10 locations (Figure A.1e).
6. Instance with four persons and 11 locations (Figure A.1f).
7. Instance with four persons and 10 locations (Figure A.1g).
8. Instance with four persons and 11 locations (Figure A.1h).
9. Instance with four persons and 12 locations (Figure A.1i).
10. Solvable instance with four persons and 10 locations (Figure A.1j).

A.2 Pegsol-Invasion

We use the following instances in the pegsol-invasion domain:

1. Instance with one filled location with 6 locations in one row (Figure A.2a).
2. Instance with four filled locations with 14 locations in two rows of 7 locations (Figure A.2b).
3. Instance with 9 filled locations with 24 locations in three rows of 8 locations (Figure A.2c).

4. Solvable instance with 5 filled locations with 24 locations in three rows of 8 locations (Figure A.2d).

We introduce for each instance two versions (the second one marked with an apostrophe). The first version marks only the initially filled, and the location which is initially free and desired to be filled as clauses of interest. The second version marks the initial and goal specification of all locations as clauses of interest.

A.3 Chessboard-Pebbling

We use the following instances in the chessboard-pebbling domain:

1. Instance with a 3×3 field with the locations l_1,l_2,l_3 initially filled and desired to be free (Figure A.3a).
2. Instance with a 4×4 field with the locations l_1,l_2,l_3 initially filled and desired to be free (Figure A.3b).
3. Instance with a 5×5 field with the locations l_1,l_2,l_3 locations initially filled and desired to be free (Figure A.3c).
4. Solvable instance with a 5×5 field with the location l_1 initially filled and the locations l_1,l_2,l_3 desired to be free (Figure A.3d).

We introduce for each instance two versions (the second one marked with an apostrophe). The first version marks only the initial and goal specification of the locations l_1,l_2,l_3 as clauses of interest. The second version marks the initial and goal specification of all locations as clauses of interest.

A.4 Slidingtiles

We use the following instances in the slidingtiles domain:

1. Instance with a 2×2 field (Figure A.4a).
2. Solvable instance with a 2×2 field (Figure A.4b).

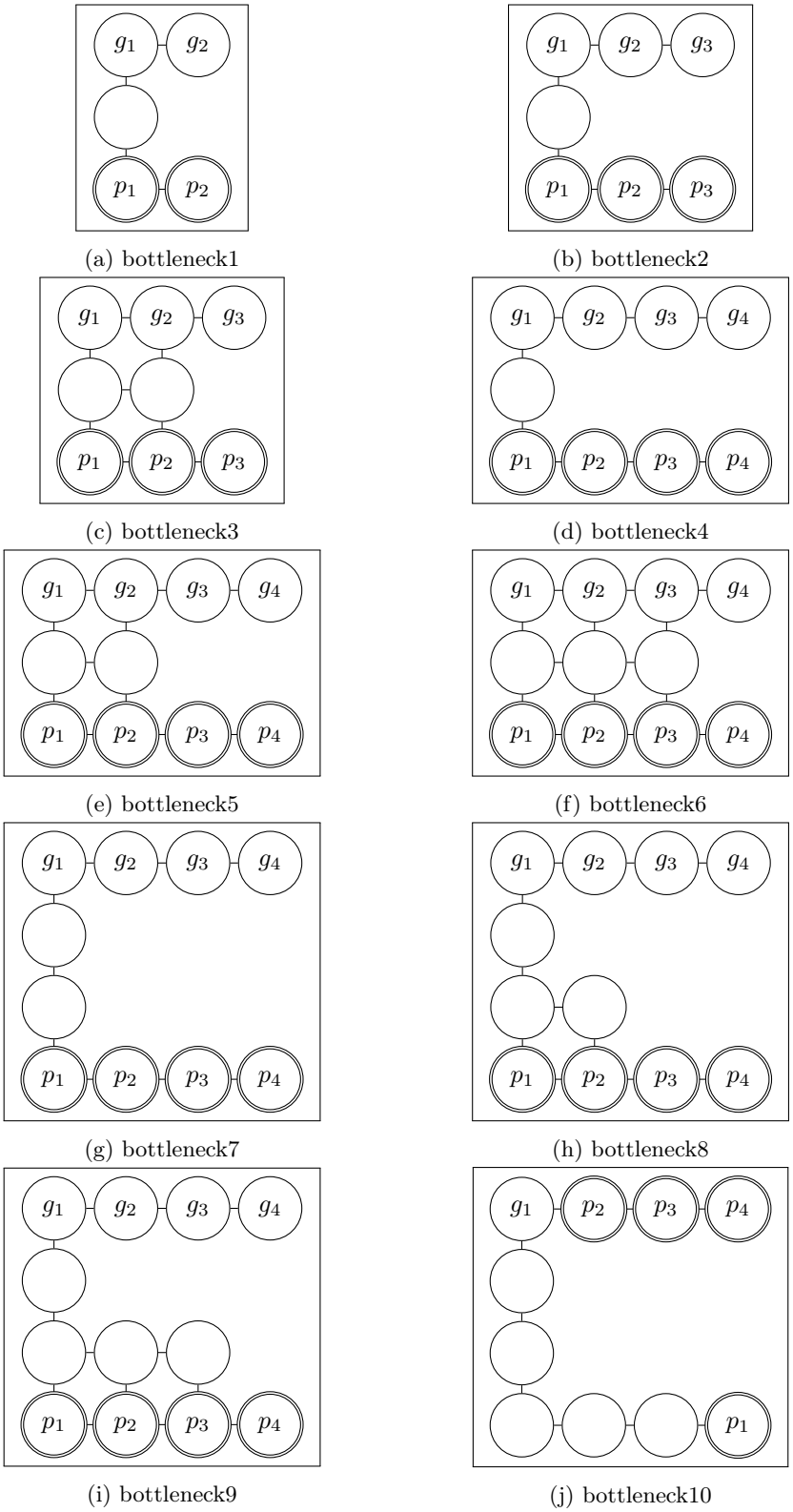
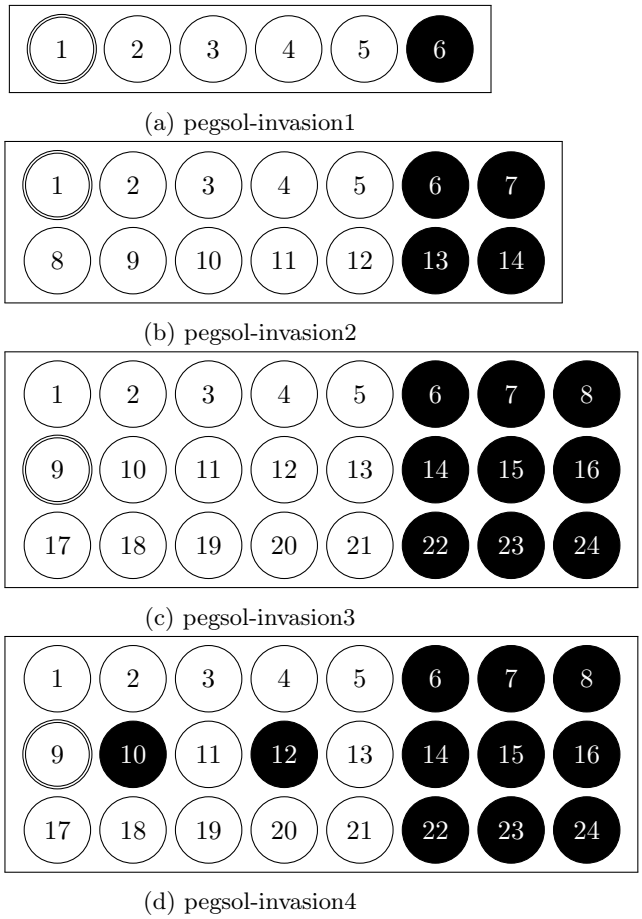


Figure A.1: Visualized bottleneck instances. Persons start in their i_p location and need to go to the g_p location. Valid connections are defined by the lines between the locations. Already visited locations are marked with a second circle.



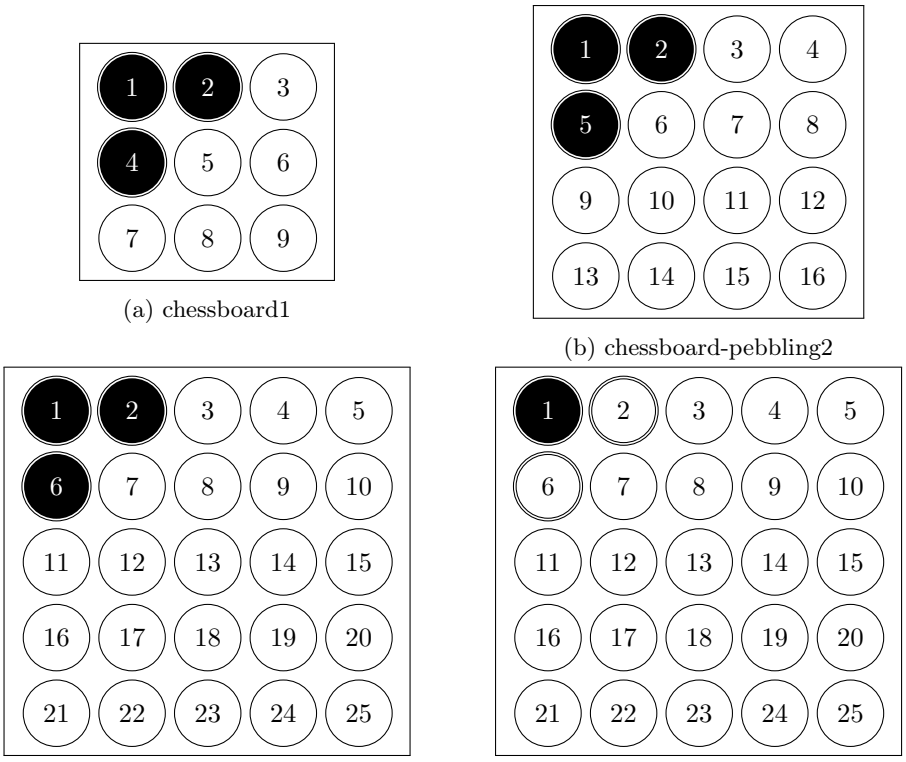
(a) pegsol-invasion1

(b) pegsol-invasion2

(c) pegsol-invasion3

(d) pegsol-invasion4

Figure A.2: Visualized pegsol-invasion instances. Locations which need to be filled are marked with a second circle. Initially filled locations are black.



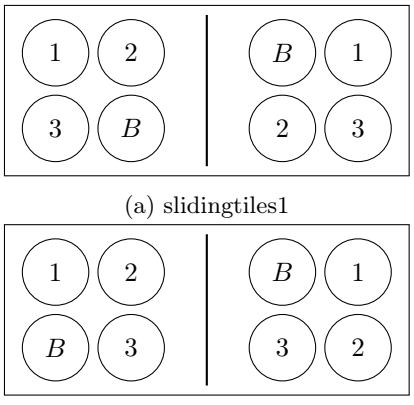
(a) chessboard1

(b) chessboard-pebbling2

(c) chessboard-pebbling3

(d) chessboard-pebbling4

Figure A.3: Visualized chessboard-pebbling instances. Locations which need to be cleared are marked with a second circle. Initially filled locations are black.



(a) slidingtiles1

(b) slidingtiles2

Figure A.4: Visualized sliding-tiles instances. The left pattern shows the initial location of tiles, and the right pattern shows the goal location of tiles.

Appendix B

Benchmark Results

We provide the results of the experiment in Section 5.2, 5.3.

B.1 Deciding Planning Tasks

We examine for each planning task several properties:

1. The amount of iterations we can reach with generating all MUSs or without (Table B.1).
2. The solving times of BUS, `PEX`, and `SymPA` (Table B.2).
3. The amount of MCSs, MUSs, clauses of interest, and actions (Table B.3).

B.2 Explaining Planning Tasks

We display for one instance of each domain the generated MCSs and MUSs. Note that we use the same instance of chessboard-pebbling as in Chapter 4 and add the clauses of interest and generated MCSs and MUSs here for completeness. The used instances are bottleneck3 (Table B.5), pegsol-invasion1 (Table B.6), chessboard-pebbling1 (Table B.7), and slidingtiles1 (Table B.8).

Definition B.2.1 (Clauses of interest for bottleneck3)**.** *The clauses of interest with the current plan horizon ph are:*

1. p_1*-initial with* $p_1^1 = 1$
2. p_2*-initial with* $p_2^1 = 2$
3. p_3*-initial with* $p_3^1 = 3$
4. p_1*-goal with* $p_1^{ph} = 7$
5. p_2*-goal with* $p_2^{ph} = 8$
6. p_3*-goal with* $p_3^{ph} = 9$

Definition B.2.2 (Clauses of interest for pegsol-invasion1)**.** *The clauses of interest with the current plan horizon ph are:*

1. l_1*-initial with* $\neg filled_1^1$
2. l_2*-initial with* $\neg filled_2^1$
3. l_3*-initial with* $\neg filled_3^1$
4. l_4*-initial with* $\neg filled_4^1$
5. l_5*-initial with* $\neg filled_5^1$
6. l_6*-initial with* $filled_6^1$
7. l_1*-goal with* $filled_1^{ph}$

Definition B.2.3 (Clauses of interest for chessboard-pebbling3)**.** *The clauses of interest with the current plan horizon ph are:*

1. l_1*-initial with* $filled_1^1$
2. l_2*-initial with* $filled_2^1$
3. l_4*-initial with* $filled_4^1$
4. l_1*-goal with* $\neg filled_1^{ph}$
5. l_2*-goal with* $\neg filled_2^{ph}$
6. l_4*-goal with* $\neg filled_4^{ph}$

Definition B.2.4 (Clauses of interest for slidingtiles1)**.** *The top-left location is marked with 1, the top-right with 2, the bottom-left with 3, and the bottom-right with 4. The clauses of interest with the current plan horizon ph are:*

1. $tileAt_1$*-initial with* $tileAt_1^1 = 1$
2. $tileAt_2$*-initial with* $tileAt_2^1 = 2$
3. $tileAt_3$*-initial with* $tileAt_3^1 = 3$
4. $tileAt_4$*-initial with* $tileAt_4^1 = 4$ *(with 4 representing the blank B)*
5. $tileAt_2$*-goal with* $tileAt_2^{ph} = 1$
6. $tileAt_3$*-goal with* $tileAt_3^{ph} = 2$
7. $tileAt_4$*-goal with* $tileAt_4^{ph} = 3$

domain	instance	$\#it_{witness}$	$\#it_{empty}$	$\#it_{MUSs}$	$\#it_{noMUSs}$
bottleneck	1	4	5	72	106
	2	5	7	23	78
	3	6	7	25	78
	4	6	-	7	63
	5	7	-	8	62
	6	8	-	8	62
	7	-	-	6	61
	8	7	-	7	63
	9	-	-	8	61
	10	-	-	6	6
pegsol-invasion	1	1	1	195	258
	2	3	3	114	159
	3	8	8	65	97
	4	-	-	3	3
	1'	1	3	172	258
	2'	3	9	27	159
	3'	-	-	5	97
chessboard-pebbling	1	5	5	241	332
	2	10	10	121	191
	3	11	11	23	24
	4	-	-	4	4
	1'	5	5	237	332
	2'	10	10	115	191
	3'	-	-	10	24
slidingtiles	1	13	13	55	94
	2	-	-	3	3

Table B.1: Results of the experiment in Section 5.2.2. For each instance we analyze in which iteration a witness is found ($\#it_{witness}$), in which iteration the empty core was generated ($\#it_{empty}$), and how many iterations can be reached with generating all MUSs ($\#it_{MUSs}$) and without ($\#it_{noMUSs}$).

domain	instance	BUS/s	PEX/s	SymPA/s
bottleneck	1	0.75	1.75	0.151
	2	3.1	11.3	0.151
	3	3	9.8	0.150
	4	21.1	-	0.148
	5	27.4	-	0.146
	6	28.1	-	0.154
	7	-	-	0.144
	8	32	-	0.148
	9	-	-	0.148
	10	1.2	2.3	0.204
pegsol-invasion	1	0.06	0.07	0.145
	2	0.35	0.61	0.153
	3	2.4	5.8	0.16
	4	0.39	0.74	0.228
	1'	0.21	1.3	0.145
	2'	1.3	-	0.153
	3'	-	-	0.16
	4'	0.42	1.1	0.228
chessboard-pebbling	1	0.96	3.3	0.147
	2	3.1	13.4	0.204
	3	5.5	21.3	0.237
	4	0.67	2.2	0.212
	1'	1.03	0.07	0.147
	2'	3.9	0.07	0.204
	3'	-	-	0.237
slidingtiles	1	6.9	25.2	0.172
	2	0.65	1.4	0.18

Table B.2: Results of the experiment in Section 5.2.2. We run BUS up to the point that the first witness is detected, PEX until the empty core was generated, and SymPA until it terminates.

domain	instance	#MCS	#MUS	*#coi*	*#action*
bottleneck	1	11	7	2+2	8
	2	30 [31]	16 [20]	3+3	12
	3	44	18	3+3	12
	4	66	30	4+4	16
	5	106	33	4+4	16
	6	153	64	4+4	16
	7	76	28	4+4	16
	8	80	29	4+4	16
	9	100	38	4+4	16
	10	10	5	4+4	16
pegsol-invasion	1	0	0	2+1	8
	2	4	2	5+1	20
	3	14	7	10+1	52
	4	4	2	10+1	52
	1'	6 [11]	8 [16]	6+1	8
	2'	22 [176]	19 [201]	14+1	20
	3'	19	11	5	52
	4'	7	2	10+1	52
chessboard-pebbling	1	31	20	3+3	4
	2	67	36	3+3	9
	3	77	42	3+3	16
	4	17	8	3+3	16
	1'	31	20	9+3	4
	2'	69	40	16+3	9
	3'	71	38	25+3	16
slidingtiles	1	87	73	4+3	8
	2	13	12	4+3	8

Table B.3: Results of the experiment in Section 5.2.2. We note how many MCSs and MUSs have been generated throughout BUS and how many clauses of interest (*#coi*, split into clauses of interest for the initial and goal configuration) and actions each instance has. The amounts of MCSs and MUSs in square brackets indicates how many `PEX` generated. This happens because BUS terminates after detecting one witness, while `PEX` continues until the empty core was generated.

instance	iteration	time	#MCS	#MUS
bottleneck8	1	0	0	0
	2	0.43	16	4
	3	0.85	32	8
	4	1.19	40	11
	5	2.13	56	15
	6	3.87	64	18
	7	10.72	74	24
pegsol-invasion3	1	0	0	0
	2	0.22	2	1
	3	0.42	4	2
	4	0.65	6	3
	5	0.9	8	4
	6	1.22	10	5
	7	1.63	12	6
	8	2.21	14	7
chessboard-pebbling3	1	0	0	0
	2	0.41	9	5
	3	0.78	21	12
	4	1.14	27	18
	5	1.42	33	20
	6	1.74	39	22
	7	2.17	47	26
	8	2.7	55	30
	9	3.35	63	34
	10	4.24	71	38
	11	5.5	77	42
slidingtiles1	1	0	0	0
	2	0.33	5	6
	3	0.58	9	10
	4	0.96	19	17
	5	1.27	24	23
	6	1.78	38	32
	7	2.36	43	37
	8	3.03	49	42
	9	3.85	54	47
	10	4.77	68	56
	11	5.43	73	62
	12	6.5	83	69
	13	7.45	87	73

Table B.4: Results of the experiment in Section 5.2.2. Progression of solving time, the amount of MCSs, and the amount of MUSs over the iterations.

plan horizon	MCSs	MUSs
1	{1,2,3},{1,2,6},{1,3,5}, {1,5,6},{2,3,4},{2,4,6}, {3,4,5},{4,5,6}	{1,4},{2,5},{3,6}
2	{1,3},{1,6},{2,3},{2,6}, {3,4},{3,5},{4,6},{5,6}	{1,2,4,5},{3,6}
3	same as 1	same as 1
4	{1,2,5},{1,3},{1,6},{2,3}, {2,4,5},{2,6},{3,4},{3,5}, {4,6},{5,6}	{1,2,4,5},{1,3,4,6}, {2,3,6},{3,5,6}
5	same as 1	same as 1
6	{1,2,5},{2,4,5}	{1,4},**{2}**,**{5}**
7	-	-

Table B.5: MCSs and MUSs generated throughout performing BUS on an unsolvable bottleneck instance. The numbers indicate which clauses appeared specified in Definition B.2.1. Bold set of clauses indicate witnesses.

plan horizon	MCSs	MUSs
1	{1,2,7},{1,3,4},{1,4,5}, {2,3},{3,4,7},{4,5,7}	{1,2,7},**{1,3,4}**,**{1,3,5}**, {1,3,7},**{2,3,5}**,**{2,4}**, {3,4,7},{3,5,7}
2	{1,2,3,5},{1,2,4,7}, {1,3,4,7},{2,3,5,7}, {2,4,5}	**{1,2}**,{1,4,7},**{1,5}**,**{2,3}**, **{2,4}**,{2,7},**{3,4}**,{5,7}
3	-	-

Table B.6: MCSs and MUSs generated throughout performing BUS on an unsolvable pegsol-invasion instance. The numbers indicate which clauses appeared specified in Definition B.2.2. Bold set of clauses indicate witnesses.

plan horizon	MCSs	MUSs
1	{1,2},{1,3},{1,5},{1,6}, {2,4},{2,3,5,6},{3,4}, {4,5},{4,6}	{1,2,4},{1,3,4},{1,4,5}, {1,4,6},{2,3,5,6}
2	{1,2},{1,3},{1,5},{1,6}, {2,4},{2,3,5},{2,3,6}, {2,5,6},{3,4},{3,5,6}, {4,5},{4,6}	{1,2,3,4},{1,2,4,5}, {1,2,4,6},{1,3,4,5}, {1,3,4,6},{1,4,5,6}, {2,3,5,6}
3	{1},{2,3,5},{2,3,6}, {2,5,6},{3,5,6},{4}	{1,2,3,4},{1,2,4,5}, {1,2,4,6},{1,3,4,5}, {1,3,4,6},{1,4,5,6}
4	{2,3},{2,5}, {2,6},{5,6}	{2,5},{3,6}
5	-	-

Table B.7: MCSs and MUSs generated throughout performing BUS on an unsolvable chessboard-pebbling instance. The numbers indicate which clauses appeared specified in Definition B.2.3.

plan horizon	MCSs	MUSs
1	{1,2,6},{2,3,4},{2,3,5}, {2,4,6,7},{5,6}	{1,3,5,7},{1,4,5},{2,5}, {2,6},{3,6},{4,5,6}
2	{1,2},{2,3},{5,6},{6,7}	{1,3,5,7},{1,3,6}, {2,5,7},{2,6}
3	{1,2,4},{1,2,5},{1,2,6}, {1,4,6},{2,3,4},{2,3,5}, {2,3,6},{2,4,7},{5,6},{6,7}	{1,2,5,7},{1,3,4,6}, {1,3,5,7},{1,3,6,7}, {2,4,5,7},{2,6},{4,5,6}
4	{1,2},{1,4},{2,3}, {5,6},{7}	{1,2,5,7},{1,2,6,7}, {1,3,5,7},{1,3,6,7}, {2,4,5,7},{2,4,6,7}
5	{1,2,4},{1,2,5},{1,3,4}, {1,3,5},{1,3,6},{1,4,5}, {1,4,6},{2,3,4},{2,3,6}, {2,4,7},{3,4,7},{5,6}, {5,7},{6,7}	{1,2,5,7},{1,2,6,7}, {1,3,5,7},{1,3,6,7}, {1,4,6,7},{2,3,4,6,7}, {2,3,5,6},{3,4,5,7}, {4,5,6}
6	{1,3},{1,4},{2,3,4,6}, {2,4,5,6},{7}	{1,2,7},{1,3,5,7},{1,4,7}, {1,6,7},{3,4,7}
7	{1,3,4},{1,3,5},{1,3,6}, {1,4,5},{3,4,7},{5,7}	{1,4,5},{1,7},{3,4,7}, {3,5},{4,5,6}
8	same as 6	same as 6
9	same as 5	same as 5
10	same as 4	same as 4
11	same as 3	same as 3
12	same as 2	same as 2
13	-	-

Table B.8: MCSs and MUSs generated throughout performing BUS on an unsolvable slidingtiles instance. The numbers indicate which clauses appeared specified in Definition B.2.4.