Extending SymPA with Unsolvability Certificates

Claudia Grundke

Bachelor Thesis

University of Basel
Departement of Mathematics and Computer Science

07 September, 2020

Abstract

Verifying whether a planning algorithm came to the correct result for a given planning task is easy if a plan is emitted which solves the problem. But if a task is unsolvable most planners just state this fact without any explanation or even proof. In this thesis we present extended versions of the symbolic search algorithms SymPA and symbolic bidirectional uniform-cost search which, if a given planning task is unsolvable, provide certificates which prove unsolvability. We also discuss a concrete implementation of this version of SymPA.

Acknowledgements

First of all I want to thank my supervisor Florian Pommerening who guided me into the right directions during the early weeks and moreover offered great support the whole time. His extensive feedback not only improved this thesis but also offered me many new insights into the topic. Next I want to thank Álvaro Torralba who gladly answered my questions about SymPA and Salomé Eriksson who gladly answered my questions regarding the verifier and even extended it to provide a specific utility. I also want to thank Malte Helmert for the opportunity to write my Bachelor thesis in his AI research group. Finally I want to thank my parents and our big black cat for all their moral support during the last three months. You are the best!

Contents

1 Introduction

In planning one aims to find a sequence of actions from an initial situation to some goal. For example take mail delivering where the initial situation is that all mail is currently at the post office and the goal is that every letter is at its destination. Possible actions in this scenario are to load mail into a delivery truck, to drive a truck to some location and to deliver a letter. This task of course should be quite feasible but imagine that the address of a letter is not readable. Then this letter cannot be delivered to its destination and the defined goal cannot be reached.

Such planning problems can be either solvable if an action sequence from start to goal exists or they can be unsolvable. Accordingly, most planning algorithms return an action sequence that solves a problem if a solution exists and declare that no solution exists if the problem is unsolvable. It is of course desirable that planning algorithms work correctly and have no bugs so that one can rely on their results. However proving correctness of an entire planning algorithm is not trivial and even more difficult the more complex an algorithm becomes. This is often the case for efficient or powerful algorithms and thus widely used algorithms. So a more feasible approach to proving correctness is needed.

Certifying algorithms (McConnell et al., 2011) address this issue. In addition to a result a certifying algorithm emits a certificate which proves that the planning algorithm came to the correct conclusion and which can be verified by an external verifier. Most planners that focus on solvable tasks are already partially certifying algorithms as the action sequence they output if a task is solvable is such a certificate. Showing that the algorithm worked properly on a given input with an action sequence as certificate is straightforward. Beginning from the initial state the actions of the sequence are applied one by one and if after applying all actions a goal was reached then the action sequence is a solution and the algorithm worked correctly on this specific input. On the other hand how to verify whether the algorithm returned a correct result if it only says the problem is unsolvable? The algorithm returns nothing which could be examined to proof that it came to the correct result and has no bugs.

Fully certifying algorithms already exist which emit certificates for both solvable and unsolvable cases. One example is the certifying version of A* with a delete-relaxation heuristic or with a merge and shrink heuristic with linear merge strategy from Eriksson (2019b). However no certifying algorithms for symbolic planners exist so far.

The goal of this thesis is to extend the planner SymPA (Torralba, 2016) to a partially certifying algorithm which provides unsolvability certificates. SymPA is a symbolic bidirectional search algorithm which uses abstractions to reduce the complexity of the planning task. It is related to the quite successful planner SymBA* (Torralba, 2015) but focuses on unsolvable problems while SymBA* concentrates on solvable tasks.

Before looking at SymPA we look at symbolic bidirectional uniform-cost search (SBU) which is already a partially certifying algorithm. SBU is very similar but less complic-

ated than SymBA* and also focuses on solvable cases. We show a version of SBU that emits a certificate for solvable and unsolvable tasks to give an example of a fully certifying symbolic planner.

2 Background

The following sections build the groundwork for symbolic planning and for proving unsolvability. In Section 2.1 we introduce the setting of classical planning and define its notation. After that we take a closer look at symbolic search in Section 2.2. Finally we finish with Section 2.3 which explains the proof system and its notation to prove unsolvability of a planning task.

2.1 Classical Planning

The idea of automated planning is that an agent can alter its environment with actions to achieve a goal. For example consider an automated vacuum cleaner as agent in a house with different rooms as its environment and possible actions for the vacuum cleaner are vacuuming the room it is currently in or moving to another room. The goal of course is that all rooms are clean. To achieve this the agent must perform a sequence of actions. This task may sound simple however there are possible instances of this setting where the goal cannot be reached and thus the problem will be unsolvable. Just think of potential obstacles that prevent the vacuum cleaner from moving to another (dusty) room.

This model of an agent capable of altering an environment via some actions can vary greatly in its complexity. In this work only classical planning, the most basic model, is considered which makes the following assumptions: firstly the environment is static meaning that only the actions of the agent can alter the environment and if it does not perform any action then the environment will not change. Secondly it is deterministic so the next state of the environment is fully determined by the current state and what action the agent will execute. Intuitively this means that every possible future state can be predicted with certainty. Finally the environment is fully observable which means that the agent knows every (relevant) aspect of the current state of the environment.

A planning problem can also be regarded as the task of finding a path in a graph. In such a graph nodes correspond to states and edges between nodes are induced by action application. This graph is called search space and can help to intuitively understand planning problems. Then a solution is a path from the node of the initial state to some goal node.

To formally describe planning problems and be able to input them to planning algorithms there exists an array of problem specification languages. For example there is STRIPS (Fikes and Nilsson, 1971) which is a very basic and early language. PDDL (Ghallab et al., 1998) is the current state-of-the-art language which also includes STRIPS

and describes planning tasks with predicate logic. In the following we will consider the SAS$^+$ formalism (Bäckström and Nebel, 1995). This is not a restriction because tasks represented in PDDL can automatically be converted into SAS$^+$ (Helmert, 2009).

Before formally defining planning tasks in SAS$^+$ we introduce the terms state and partial state.

Definition 1. *(state and partial state) Given a set of variables V a partial state is a partial function $s : V \to \bigcup_{v \in V} dom(v)$ that assigns variables a value from their corresponding domain. A state is a total function that assigns all variables in V.*

Variables assigned by a partial state s_p are described as vars(s_p) thus for a state s_t we have vars$(s_t) = V$.

Partial states (and partial functions) are interpreted as sets of pairs of state variables and values from their domain. For example given

the set of state variables $V = \{x, y, z\}$ with

the domains $dom(x) = dom(y) = dom(z) = \{apple, cherry, peach\}$

a possible partial state is $s_p = \{x \to cherry, z \to peach\}$

with vars$(s_p) = \{x, z\}$ and

a possible state is $s_t = \{x \to peach, y \to apple, z \to apple\}$.

Both s_p and s_t are partial states but while (total) state s_t is a special case of a partial state and only describes one state, partial state s_p can actually describe a set of (total) states. It represents all states where $x = cherry$, $z = peach$ and y has any possible value.

A SAS$^+$ task is then defined as follows.

Definition 2. *(SAS$^+$ planning task) A SAS$^+$ planning task is defined as the tuple $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$, where*

- *V^Π is a finite set of state variables with finite domain $dom(v)$ for all $v \in V^\Pi$,*
- *A^Π is a set of actions a = $\langle$pre(a), eff(a)$\rangle$ where pre(a) and eff(a) are partial states,*
- *I^Π is the inital state,*
- *G^Π is a partial state specifying all goal states.*

An action a of the set of actions A^Π could also be defined as a three tuple instead of as a pair. Such a three tuple includes next to $pre(a)$ and *eff*(a) also the cost of an action. But as action costs do not influence whether a task is solvable or not they are not included in this definition. $pre(a)$ and *eff*(a) of an action a both describe sets of states. $pre(a)$ represents

all states that fulfill the preconditions of action a and thus all states in which action a can be applied to reach a new state. *eff(a)* represents all these possible new states and describes the effects of applying action a to a state that is consistent with $pre(a)$.

Action application enables to traverse from one state of the world to another state. In the following action application in forward direction (progression) as well as action application in backward direction (regression) is defined. While the first is reasonable as one tries to reach a goal state from the initial state one might wonder why backward application is mentioned here. Trying to reach the initial state from any goal state sounds like the complete opposite of the problem. However bidirectional search uses regression as well as progression and is a quite successful approach as can be seen in Section 3.

Definition 3. *(progression) Given a planning task Π, an action $a \in A^{\Pi}$ is applicable in progression in state s if $pre(a) \subseteq s$. Then*

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

is the successor state of s.

An action a is applicable in progression in a state s if this state is consistent with $pre(a)$. Intuitively this means that an action is applicable if state s meets all preconditions. Then all variables mentioned by the effects of action a are assigned according to eff(a) and all other variables remain unchanged.

Definition 4. *(regression) Given a planning task Π, an action $a \in A^{\Pi}$ is applicable in regression in partial state s if*

- *a is relevant to s, vars$(s) \cap$ vars(eff$(a)) \neq \emptyset$ and*
- *$\forall v \in$ vars$(s) : (v \notin$ vars(eff$(a))$ or $s(v) =$ eff$(a)(v))$ and*
- *$\forall v \in$ vars$(s) : (v \notin$ vars(prv$(a))$ or $s(v) =$ prv$(a)(v))$ with $prv(a) = \{v \mapsto d \mid (v \mapsto d) \in$ pre(a) and $v \notin$ vars(eff$(a))\}$.*

The partial predecessor state of s is then defined as follows:

$$([a]s)(v) = \begin{cases} \textit{pre}(a)(v), & \textit{if } v \in \textit{vars}(\textit{pre}(a)) \\ s(v), & \textit{if } v \notin \textit{vars}(\textit{pre}(a)) \textit{ and } v \notin \textit{vars}(\textit{eff}(a)) \end{cases}$$

To apply an action a in regression in partial state s three conditions must be met. First the action has to be relevant to s otherwise all actions could be backwardly applied to the empty set (which is technically a partial state) which is not helpful in solving a planning

task. Secondly s must comply with eff(a) which means that all variables of s are either assigned the same value as described in eff(a) or they are not mentioned in eff(a). Finally s must comply with all prevailing variables which are all variables that appear in pre(a) but not in eff(a). If these requirements are fulfilled action a can be applied in regression in s. All variables of partial state $[a]s$ that are mentioned by pre(a) are assigned according to pre(a) and all variables that are neither mentioned in pre(a) nor in eff(a) are assigned the same as in s.

Comparing these two definitions it can be seen that backward action application looks quite different and more complicated than forward action application. One reason for that is that regression is defined for partial states rather than only for states. That is the case because later regression will be used to obtain the predecessors of all goal states which are represented as a partial state G^{Π}.

Now that action application is defined we can specify what a solution of a planning task looks like.

Definition 5. *(plan) Given a planning task* Π*, an* s*-plan solves state* s *and is a sequence of actions* $\pi = \langle a_1, ..., a_n \rangle$ *where* a_1 *is applicable in* s*,* a_2 *is applicable in* $s[a_1]$ *and so forth and* $G^{\Pi} \subseteq s[\pi] = ((s[a_1])...)[a_n]$*. The length of an* s*-plan is* $|\pi| = n$*.*

The I^{Π}*-plan is just called plan and solves the whole planning task.*

To find a plan most planning algorithms maintain an open and a close list. The open list is usually initialized with the initial state and stores all states that still have to be expanded. The closed list on the other hand stores the states that have already been expanded. Expanding a state means generating its successors for all actions applicable in the state. Search algorithms basically expand the states in the open list one by one until a goal state is found which means the task is solvable or until the open list becomes empty without finding a goal state which means that the task is unsolvable. In case a goal state was found the action sequence from the initial state to the goal state is reconstructed and returned as solution.

2.2 Symbolic Search

While explicit search looks at each state separately symbolic search works with sets of states. To manipulate sets of states efficiently appropriate data structures are needed. In our case Binary Decision Diagrams (BDDs) are used (Bryant, 1986). BDDs are rooted, directed, acyclic graphs that represent logical formulas. A BDD includes terminal nodes which are either associated with "true" or with "false" and decision nodes which are associated with the variables of the formula it represents. Although usually the term BDD is used it often actually refers to the term Reduced Ordered Binary Decision Diagram (ROBDD). We will keep this notion and use BDD as a synonym for ROBDD. A ROBDD is unique for a given formula and variable order compared to a BDD.

Using BDDs states are interpreted as logical formulas where the variables of the state are also used in the formula. The formula then encodes which values are assigned to which variables such that it evaluates to true only if all variables are assigned the same as in the state it represents. When representing a set of states the formula includes all variables of all states and evaluates to true only if the state belongs to the set. To represent SAS^+ states with BDDs each variable v with finite domain D_v can be represented by $\lceil \log_2 |D_v| \rceil$ binary variables in the BDD.

As we are going to work with sets of states we will now define action application for sets of states and at the same time for sets of actions using the Definitions 3.8 and 3.9 from Eriksson (2019a).

Definition 6. *(progression for sets) Given a planning task Π, state set $S \subseteq S^\Pi$ and action $a \in A^\Pi$, $S[a] = \{s[a] \mid s \in S, a \text{ applicable in } s\}$ is the progression of S with a.*

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

The progression of a state set S with action set A calculates all successors that can be reached by a state in S. The regression of a state set S with action set A on the other hand calculates all predecessors from which a state in S can be reached.

Definition 7. *(regression for sets) Given a planning task Π, state set $S \subseteq S^\Pi$ and action $a \in A^\Pi$, $[a]S = \{s' \mid a \text{ applicable in } s', s'[a] \in S\}$ is the regression of S with a.*

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

Using these action applications for sets of states symbolic search algorithms can expand not only single states but also state sets. This can mean a significant increase of performance while preserving the core ideas of many explicit search algorithms. Especially for showing unsolvability of a planning task this is very useful because the whole search space must be exhaustively explored for this.

2.3 Unsolvability Proof System

Providing a certificate for solvability is usually no problem because most planners emit a plan if the given task is solvable. Applying this plan then can verify whether the planner came to the correct conclusion for the given problem. In the following the proof system from Eriksson (2019a) is presented as a way to provide certificates for unsolvability. The certificates for unsolvability of SBU and SymPA are based on this proof system.

The general idea of showing unsolvability with the proof system is to prove that the initial state I^Π or the goal states G^Π of a planning task Π are dead. Deadness of a state or a set of states is defined in the following using Definition 5.1 from Eriksson (2019a).

Definition 8. *(dead state and dead state set) A state s is dead if no plan traverses s, or more precisely there is no plan $\pi = \langle a_1, ..., a_n \rangle$ and $0 \leq i \leq n$ with $s = I[a_1]...[a_i]$. A set of states is dead if all its elements are dead.*

Showing that the initial state or all goal states are dead then means that the task is unsolvable because an action sequence must traverse the initial state and at least one goal state to be a plan.

Regarding notation used by the components of the proof system we keep the grammar which is used by Eriksson (2019a):

state set variables $X := \{I^\Pi\} \mid S_G^\Pi \mid \emptyset \mid X_\mathbf{R}$

state set literals $L := X \mid \overline{X}$

state set expressions $S := L \mid (S \cup S) \mid (S \cap S) \mid S[A] \mid [A]S$

action set expressions $A := A^\Pi \mid a \mid (A \cup A)$

set expressions $E := S \mid A$

A state set variable X describes either the set containing only the initial state $\{I^\Pi\}$, the set of all goal states S_G^Π, the empty set $\emptyset$ or $X_\mathbf{R}$, a state set described by formalism **R**. We do not consider different formalisms so **R** will always stand for BDDs. An exception to this would be that BDDs with different variable orders are considered to be different formalisms, however, within the proofs of this thesis all used BDDs have the same variable order.

An action set expression A represents the whole set of actions A^Π of planning task Π, a single action $a \in A^\Pi$ or the union of two action sets $(A \cup A)$.

Now follow the derivation rules that can be used by proofs within the unsolvability proof system to derive judgements. To emphasize that judgements are interpreted on a purely syntactical level $S \sqsubseteq S$ is used instead of $S \subseteq S$.

Derivation Rules

Empty **D**ead $\dfrac{}{\emptyset \text{ dead}}$ ED

Union **D**ead $\dfrac{S \text{ dead} \qquad S' \text{ dead}}{S \cup S' \text{ dead}}$ UD

Subset **D**ead $\dfrac{S' \text{ dead} \qquad S \sqsubseteq S' \text{ dead}}{S \text{ dead}}$ SD

Progression **G**oal $\dfrac{S[A^{\Pi}] \sqsubseteq S \cup S' \qquad S' \text{ dead} \qquad S \cap S^{\Pi}_G \text{ dead}}{S \text{ dead}}$ PG

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

Regression **G**oal $\dfrac{[A^{\Pi}]S \sqsubseteq S \cup S' \qquad S' \text{ dead} \qquad \overline{S} \cap S^{\Pi}_G \text{ dead}}{\overline{S} \text{ dead}}$ PG

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

Conclusion **I**nitial $\dfrac{\{I^{\Pi}\} \text{ dead}}{\text{unsolvable}}$ CI

Conclusion **G**oal $\dfrac{S^{\Pi}_G \text{ dead}}{\text{unsolvable}}$ CG

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

Union **L**eft $\dfrac{}{E \sqsubseteq (E' \cup E)}$ UR

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

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

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

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

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

Subset **T**ransitivity $\dfrac{E \sqsubseteq E' \qquad E' \sqsubseteq E''}{E \sqsubseteq E''}$ SI

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

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

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

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

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

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

These inference rules do not look at the actual semantics of the sets they involve and depend on judgements that already exist about the sets. So we still need a starting point from which new judgements can be derived with these rules. To solve this problem an additional source of judgements is defined in the following that takes the semantics of the sets into account. These so called basic statements cannot be proven within the proof system and must be proven separately for every proof.

Basic Statements

B1 $\bigcap_{L \in \mathcal{L}} L \subseteq \bigcup_{L' \in \mathcal{L}'} L'$

B2 $(\bigcap_{X \in \mathcal{X}} X)[A] \cap \bigcap_{L \in \mathcal{L}} L \subseteq \bigcup_{L' \in \mathcal{L}'} L'$

B3 $[A](\bigcap_{X \in \mathcal{X}} X) \cap \bigcap_{L \in \mathcal{L}} L \subseteq \bigcup_{L' \in \mathcal{L}'} L'$

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

B5 $A \subseteq A'$

With **B1** knowledge about two state sets can be described. It must be stated as a finite intersection of state set literals being a subset of a finite union of other state set literals. Both the intersection and the union of course could contain only a single element.

To describe knowledge about a progression **B2** is used. It is similar to **B1** but includes the progression of an intersection of state set variables on the left side. Accordingly **B3** states knowledge about a regression of an intersection of state set variables.

$L_{\mathbf{R}}$ and $L'_{\mathbf{R}'}$ in **B4** represent state set literals of two different formalisms $\mathbf{R}$ and $\mathbf{R}'$. However as we will not consider different formalisms not even BDDs with different variable orders **B4** is not relevant for the following sections.

B5 can be used to state knowledge about different action sets A and A' but is not needed for the proofs of this thesis.

3 Certifying Symbolic Bidirectional Uniform-Cost Search

Uniform-cost search, also known as Dijkstra's algorithm, is an algorithm to find shortest paths in graphs (Dijkstra, 1959). It expands the states with smallest g-value first where g is the total cost of all actions that have to be applied to get from the initial state to the current state. Prioritizing small g-values ensures that the found paths to all expanded states are optimal.

Symbolic uniform-cost search expands all states with smallest g-value at once instead of one state after another. It represents sets of states with BDDs and operates on them.

Symbolic bidirectional uniform-cost search (SBU for short) interleaves two symbolic uniform-cost searches. The forward search starts at the initial state and performs progression toward the goal states and the backward search begins at the set of goal states and performs regression toward the initial state. When new states are generated it is checked whether the search of the other direction already reached these states. If this is the case a solution is found. However to find the optimal (that means cheapest) plan the search has to be continued until the summed g-values of forward and backward search exceed the total cost of the best known plan.

Certifying symbolic bidirectional uniform-cost search is the same as symbolic bidirectional uniform-cost search with the extension of generating unsolvability certificates if no solution was found. In the following section the certifying SBU algorithm is presented and in Section 3.2 the unsolvability proofs generated by this algorithm will be shown.

3.1 Algorithm

Algorithm 1 shows the pseudo code of certifying SBU which is based on SBU from Torralba (2015). In addition to the planning task Π it takes the transition relations $\mathcal{T}$ as input which is needed to expand states. $\mathcal{T}$ is a set of transition relations T_c. For each action cost c there exists the transition relation T_c which contains state tuples (s, s') such that s' is the successor of s along an action with cost c. The algorithm begins with the initialization of the open lists $fOpen$ and $bOpen$ and the closed lists $fClosed$ and $bClosed$ of the forward and backward directions. They are all sorted by g-value and store states with the same g-values as one BDD. The closed lists are initialized as empty BDDs while the forward open list $fOpen$ begins with the initial state at $g = 0$ and the backward closed list $bOpen$ begins with the set of goal states at $g = 0$. g_f and g_b are initialized with 0 and hold the costs needed to reach the currently most promising states by forward or backward search respectively. w_{total} is initialized with ∞ and keeps the total costs of the current best plan if a plan was found.

Algorithm 1: Certifying Symbolic Bidirectional Uniform-Cost Search

Input: Planning Problem: $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$
Input: Transition relations: $\mathcal{T}$
Output: Cost-optimal plan or certificate for unsolvability

$fOpen_0 \leftarrow \{I^\Pi\}$
$bOpen_0 \leftarrow G^\Pi$
$fClosed \leftarrow bClosed \leftarrow \bot$
$g_f \leftarrow g_b \leftarrow 0$
$w_{total} \leftarrow \infty$
$\pi \leftarrow$ "no plan"
while $g_f + g_b < w_{total}$ **and not** $fOpen$ *is empty* **and not** $bOpen$ is empty **do**
    if `NextStepDirection`*($fOpen$, $bOpen$)* = *Forward* **then**
        $fOpen, fClosed, g_f, \pi, w_{total} \leftarrow$ `Step`$(fOpen, fClosed, g_f, bClosed, \mathcal{T}, \pi, w_{total})$
    else
        $bOpen, bClosed, g_b, \pi, w_{total} \leftarrow$ `Step`$(bOpen, bClosed, g_b, fClosed, \mathcal{T}^{-1}, \pi, w_{total})$
if $w_{total} < \infty$ **then**
    return π
else
    if $fOpen$ *is empty* **then**
        return `GenerateForwardCertificate`$(fClosed_*, I^\Pi, G^\Pi)$
    else
        return `GenerateBackwardCertificate`$(bClosed_*, I^\Pi, G^\Pi)$
Procedure `Step`$(open, closed, g_{min}, closed', \mathcal{T}, \pi, w_{total})$
    $open_{g_{min}} \leftarrow$ `BFS`$(open_{g_{min}}, T_0, closed_*, \emptyset)$
    $closed_{g_{min}} \leftarrow open_{g_{min}}$
    $\pi, w_{total} \leftarrow$ `UpdatePlan`$(\pi, w_{total}, open_{g_{min}}, g_{min}, closed')$
    for all $T_c \in \mathcal{T}, c > 0$ **do**
        if $g_{min} + c < w_{total}$ **then**
            $Succ \leftarrow image(open_{g_{min}}, T_c) \wedge \neg closed_*$
            $\pi, w_{total} \leftarrow$ `UpdatePlan`$(\pi, w_{total}, Succ, g_{min} + c, closed')$
            $open_{g_{min}+c} \leftarrow open_{g_{min}+c} \vee (Succ \wedge \neg closed'_*)$
    $open_{g_{min}} \leftarrow \bot$
    $g_{min} \leftarrow \min\{g \mid open_g \neq \bot\}$
    return $open, closed, g_{min}, \pi, w_{total}$

The main loop of SBU continues until the sum of g_f and g_b becomes greater or equal than w_{total} or until one of the open lists becomes empty (line 7). The former means that the sum of the path costs to the currently most promising states in both directions is greater than the cost of the currently known best plan. Because g_f and g_b can only increase this means that no better plan can be found and the algorithm can return the current plan as an optimal plan. If the loop breaks because an open list becomes empty this means that one search exhaustively explored the search space so the search of the other direction cannot contribute further and the algorithm can terminate.

In every iteration one search performs a search step. The decision whether forward or backward search is continued depends on which direction looks more promising which is mostly based on a time estimation. The `Step` procedure begins with a breadth first search to find all states that are reachable with no costs from the current set of states in the open list with minimal path cost g_{min}. The found states are added to the states in the open list that currently have the lowest path cost (line 20). These states with currently lowest path cost are added to the closed list, used to update the plan and then expanded. The `UpdatePlan` procedure compares these states with the closed list of the search of the opposite direction and checks whether they intersect (line 22). In that case a plan was found and if this plan is cheaper than the current plan π or if no plan was found until now then π is updated to hold the new plan and w_{total} is updated to hold its cost. After updating the plan the states with currently lowest path cost are expanded. For every transition with action cost $c > 0$ the successors of the current states are generated (line 25) if the cost of applying the action of this transition does not exceed the path cost of the current best plan (line 24). If successors were generated then the plan is updated again and the successors are put into the open list according to their g-value (line 27). After expanding the states with currently lowest g-value they are removed from the open list and the new minimal g-value is computed (line 29). This value will be greater or equal than the last because the states with the last minimal g-value were removed from the open list and the states that were added to the open list are successors of these states and have greater g-values because actions with action costs greater zero were applied to reach them.

When the main loop terminates w_{total} is used to check whether a solution was found. This check in line 12 and the following lines until line 18 are the extension such that SBU becomes certifying SBU. Instead of this check and the certificate generation SBU just returns π at this point not matter if it contains a plan or is still set to "no plan". If w_{total} is smaller than ∞ then a solution was found because w_{total} was updated at some point with the cost of a plan. Then this plan π can be returned. Otherwise no solution was found and a certificate for unsolvability is generated. Depending on which open list became empty a certificate for the forward direction or for the backward direction is generated. If the forward open list $fOpen$ became empty then the forward search expanded all reachable states without finding a plan and a certificate based on that is generated otherwise the same

#	judgement	rule	premises
(1)	$\emptyset$ dead	**ED**	
(2)	$fClosed_*[A^\Pi] \sqsubseteq fClosed_*$	**B2**	
(3)	$fClosed_* \sqsubseteq fClosed_* \cup \emptyset$	**UR**	
(4)	$fClosed_*[A^\Pi] \sqsubseteq fClosed_* \cup \emptyset$	**ST**	(2), (3)
(5)	$fClosed_* \cap G^\Pi \sqsubseteq \emptyset$	**B1**	
(6)	$fClosed_* \cap G^\Pi$ dead	**SD**	(1), (5)
(7)	$fClosed_*$ dead	**PG**	(4), (1), (6)
(8)	$\{I^\Pi\} \sqsubseteq fClosed_*$	**B1**	
(9)	$\{I^\Pi\}$ dead	**SD**	(7), (8)
(10)	task unsolvable	**CI**	(9)

Figure 1: SBU unsolvability proof in forward direction ($fOpen = \emptyset$)

goes for the backward direction. In the following section we take a closer look at these proofs.

3.2 Unsolvability certificates

To show unsolvability with the proof system it must be proven that the initial state or all goal states are dead. This is done mostly on a purely syntactical level and can be regarded for SBU in Figures 1 and 2. High-level explanations follow shortly. A second part of unsolvability proofs are basic statements which build the basis for syntactical arguments and are just stated and then used in Figure 1 and Figure 2. These basic statements have to be checked by the verifier separately from the proof system for each certificate in practice but in the following it will be explained why they hold in general if the problem is unsolvable.

Certifying unsolvability of SBU happens in one of two different ways depending on whether the forward open list $fOpen$ or the backward open list $bOpen$ becomes empty. The idea behind both unsolvability proofs is to show that the set of states created by the closed list cannot be left and is dead (it contains no plan).

The proof in forward direction (Figure 1) shows unsolvability by proving that the initial state is dead. The first considered basic statement claims that the set of states $fClosed_*$ cannot be left by progression (judgement 2). Where $fClosed_* = \bigvee_i fClosed_i$ is the disjunction of all BDDs in the forward closed list. In other words this states that all successors of states in $fClosed_*$ are also in $fClosed_*$. This is true if $fOpen$ is empty because during SBU first the initial state is inserted into $fOpen$ and when one entry in $fOpen$ is removed it is inserted into $fClosed$ and all its successors are inserted into $fOpen$. So if $fOpen$ becomes empty (which is a requirement to generate a certificate as can be seen in line 16 of Algorithm 1) all states reachable from the initial state have been inserted into $fClosed$.

#	judgement	rule	premises
(1)	$\emptyset$ dead	**ED**	
(2)	$[A^{\Pi}]bClosed_* \sqsubseteq bClosed_*$	**B3**	
(3)	$bClosed_* \sqsubseteq bClosed_* \cup \emptyset$	**UR**	
(4)	$[A^{\Pi}]bClosed_* \sqsubseteq bClosed_* \cup \emptyset$	**ST**	(2), (3)
(5)	$\{I^{\Pi}\} \sqsubseteq \overline{bClosed_*}$	**B1**	
(6)	$bClosed_*$ dead	**RI**	(4), (1), (5)
(7)	$G^{\Pi} \sqsubseteq bClosed_*$	**B1**	
(8)	G^{Π} dead	**SD**	(6), (7)
(9)	task unsolvable	**CG**	(8)

Figure 2: SBU unsolvability proof in backward direction ($bOpen = \emptyset$)

They are still contained in it at the end of the algorithm because the algorithm does not remove anything from $fClosed$.

The second basic statement used in the proof says that $fClosed_*$ contains no goal state (judgement 5). For syntactical reasons this is stated as the intersection of $fClosed_*$ and the goal states S_G^{Π} being a subset of the empty set. This basic statement is true because $fClosed_*$ contains all states reachable from the initial state and if any goal state were reachable from $\{I^{\Pi}\}$ then the task would be solvable and would return a plan instead of the unsolvability certificate.

From these two facts that $fClosed_*$ cannot be left (judgement 2) and that $fClosed_*$ contains no goal state (judgement 5) it follows (via a syntactical detour) that $fClosed_*$ is dead (judgement 7) which makes sense because if $fClosed_*$ cannot be left and contains the initial state but no goal state then no plan can ever traverse it.

The last basic statement tells that the initial state is in $fClosed_*$ (judgement 8) which is true because at the very beginning of Algorithm 1 it is inserted into $fOpen$ later removed from $fOpen$ and inserted into $fClosed$ (line 21) and never removed from $fClosed$. So because $fClosed_*$ is dead it follows that the initial state is dead (judgement 9) and thus that the task is unsolvable (judgement 10). This concludes the proof because the initial state cannot be part of a plan.

The proof in backward direction (Figure 2) shows that all goal states are dead contrary to the forward direction where deadness of the initial state was proven. Despite that opposite aim it works very similar to the proof in forward direction. Its first basic statement says that $bClosed_*$ cannot be left by regression (judgement 2) and whose only differences to the first basic statement in forward direction are that $bClosed_*$ is used instead of $fClosed_*$ and regression instead of progression. Therefore $bClosed_* = \bigvee_i bClosed_i$ is defined analogously to $fClosed_*$ as the disjunction of all BDDs in the backward closed list and also the argument why all predecessors of $bClosed_*$ are contained in $bClosed_*$ works accord-

ingly: beginning from the set of goal states all backward reachable states are inserted into $bClosed$ during Algorithm 1 and none are removed from $bClosed$.

The next basic statement looks quite different to its counterpart in forward direction but has an analogous meaning. It states that the initial state is not in $bClosed_*$ (again for syntactical reasons) written as $\{I^\Pi\}$ being in the complement of $bClosed_*$ (judgement 5) and it is true because if $\{I^\Pi\}$ were in $bClosed_*$ the task would be solvable as $bClosed_*$ contains all states backward reachable from the goal states and thus a goal state would be (forward) reachable from the initial state.

Almost identical to the step in forward direction the derivation that $bClosed_*$ is dead (judgement 6) follows from that $bClosed_*$ cannot be left by regression (judgement 2) and that the initial state is not in $bClosed_*$ (judgement 5). This holds because all states backward reachable from any goal state are in $bClosed_*$ but the initial state is not in this set therefore no plan can traverse $bClosed_*$ and it is dead.

The third basic statement claims that all goal states are in $bClosed_*$ (judgement 7) which is true because the set of goal states is at the beginning of Algorithm 1 put into $bOpen$ and later into $bClosed$ (because all entries from $bOpen$ must have been inserted into $bClosed$ if $bOpen$ is empty) from where it is never removed. From this it follows directly that all goal states are dead (judgement 8) and thus that the task is unsolvable (judgement 9) because no goal can be part of a plan.

4 Certifying SymPA

The name SymPA stands for symbolic perimeter abstractions and just like symbolic bidirectional uniform-cost search SymPA is a symbolic bidirectional search algorithm. However it uses breadth first searches instead of uniform-cost searches and makes use of perimeter abstractions. The focus of SymPA is to show unsolvability as fast as possible. Thus aspects like action costs are neglected which improves the algorithm's efficiency in most cases.

The idea of SymPA is to start a forward or backward search in the original search space depending on which directions looks more promising and as soon as this search becomes too costly a new forward or backward search is started in an abstract search space where fewer states have to be considered. This strategy is not helpful for showing solvability because a problem could be solvable in an abstract search space but not in the original search space. For showing unsolvability however this is no problem at all because a task which is unsolvable in an abstract search space is also unsolvable in the original search space. Even if such a search in an abstract state space does not show unsolvability it can simplify the original search in the other direction by removing unreachable states from the open list of the other direction.

In the following a description of the certifying version of SymPA is given and then in Section 4.2 we explain how unsolvability is proven with SymPA.

4.1 Algorithm

Given a planning task Π certifying SymPA detects whether Π is solvable and in case the problem is deemed not solvable certifying SymPA provides a proof which can be verified by an external verifier (Eriksson, 2019b).

Firstly regarding notation of Algorithm 2, $\mathcal{T}_u^X$ is a breadth first search in search space X in direction u and $\mathcal{T}_{\neg u}^X$ is a breadth first search in the same search space but in opposite direction. Search direction u is either *fw* or bw and accordingly $\neg u$ is then the other one. X represents either the original search space ($X = \Pi$) or it describes an abstract search space ($X \neq \Pi$).

Every search $\mathcal{T}_u^X$ consists of an open list $\mathcal{T}_u^X.open$ which stores the states of the current search frontier that are not yet expanded and a closed list $\mathcal{T}_u^X.closed$ which stores the states that have already been expanded. Both lists are sets of states and are each stored as a single BDD.

Certifying SymPA maintains a set of ongoing searches, $SearchPool$ which is initialized with the forward and backward searches in the original state space. The forward search $\mathcal{T}_{fw}^{\Pi}$ begins with the set containing only the initial state and performs progression in every step. The backward search $\mathcal{T}_{bw}^{\Pi}$ begins with the set of all goal states and performs regression in every step.

Furthermore D_{fw} and D_{bw} the sets of dead ends that were found by finished searches are maintained for both directions and are initialized with the empty set. The open list as well as the closed list of a search are each represented by a single BDD. The found dead end states are pruned from all searches of the corresponding direction to speed up the following search steps.

Until the problem is decided to be solvable or unsolvable the algorithm loops. First in every iteration it is examined whether $SearchPool$ contains a search $\mathcal{T}_u^X$ that is feasible. A search is considered feasible if its search frontier consists of less than M states. M is a parameter of the algorithm (not mentioned in the pseudocode of Algorithm 2) and is adjusted dynamically. Beginning with a small M few states are allowed in the frontier so abstract searches are favored, increasing M lets the algorithm perform searches in less relaxed state spaces. Initialising M with infinity thus causes the algorithm to never start searches in abstract state spaces because $\mathcal{T}_{fw}^{\Pi}$ and $\mathcal{T}_{bw}^{\Pi}$ are always deemed feasible.

If $SearchPool$ contains feasible searches the easiest (according to a time estimation based on previous steps and the number of nodes representing the frontier) is chosen and it performs one step, expanding the search frontier (line 6). This means that the states of the current frontier are moved from the open list to the closed list and all their successors except those that are already known to be dead ends are put into the open list as the next frontier.

Otherwise if $SearchPool$ does not contain any feasible searches then one of the searches in the original search space is randomly selected as current search $\mathcal{T}_u^X$ and removed from

the pool of searches (line 8). While $\mathcal{T}_u^X$ is not finished it either performs one search step if it is a feasible search (which is of course not the case in the first iteration because no feasible searches are in $SearchPool$ and thus the searches in the original state space are also not feasible) or $\mathcal{T}_u^X$ is put back into $SearchPool$ to be continued later if it becomes feasible. In this case the current frontier of $\mathcal{T}_u^X$ is relaxed (line 14) to create a new current search $\mathcal{T}_u^X$ in a new abstract search space $X \neq \Pi$.

For relaxation symbolic perimeter pattern databases are used. Roughly speaking with this abstraction strategy states that share the same values for some of the variables are considered one state although they might have different values for the other variables. This reduces the amount of states the search has to consider when expanding the search frontier. For more details about the abstractions used in SymPA we refer to Torralba (2016).

After performing a step of any search or finishing a space in an abstract space it is checked whether the algorithm can terminate. If the current search $\mathcal{T}_u^X$ is finished and no solution was found then the task is unsolvable and a certificate for unsolvability can be returned. Depending on the direction of the current search a different proof is generated and emitted (line 18 and line 20).

If the current search is finished, found a solution and is in the original search space the task is solvable and the algorithm terminates with "Solvable" as its result (line 22).

Otherwise the current search is finished and found a solution but is in an abstract state space. So the task is solvable in this simplified search space X but we do not know yet whether it is solvable in the original search space. Therefore the current finished search is removed from $SearchPool$ (line 23) and all states it did not reach are put into the set of dead ends of the opposite direction $D_{\neg u}$ (line 24). Then these new dead ends are removed from all searches in direction $\neg u$ (line 25). Doing this does not remove any state which could be part of a plan from the searches. All abstract states reachable by the current search $\mathcal{T}_u^X$ were collected and this set of states must contain the initial state and at least one goal state because a plan was found. Thus all states that could be part of a plan are collected this way which means on the other hand that all states not collected this way cannot be part of a plan.

Algorithm 2: Certifying SymPA

Input: Planning Problem: $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$
Output: "Solvable" or certificate for unsolvability

1 $SearchPool \leftarrow \{\mathcal{T}^\Pi_{fw}, \mathcal{T}^\Pi_{bw}\}$
2 $D_{fw}, D_{bw} \leftarrow \emptyset, \emptyset$
3 **Loop**
4 **if** $\exists \mathcal{T}^X_u \in SearchPool$ *s.t.* `IsFeasible`$(\mathcal{T}^X_u)$ **then**
5 $\mathcal{T}^X_u \leftarrow$ `Easiest-Search`$(SearchPool)$
6 `ExpandFrontier`$(\mathcal{T}^X_u, D_u)$
7 **else**
8 $\mathcal{T}^X_u \leftarrow$`RandomSelection`$(\mathcal{T}^\Pi_{fw}, \mathcal{T}^\Pi_{bw})$
9 **while** $\mathcal{T}^X_u$ *is not finished* **do**
10 **if** `IsFeasible`$(\mathcal{T}^X_u)$ **then**
11 `ExpandFrontier`$(\mathcal{T}^X_u, D_u)$
12 **else**
13 $SearchPool \leftarrow SearchPool \cup \{\mathcal{T}^X_u\}$
14 $\mathcal{T}^X_u \leftarrow$ `RelaxFrontier`$(\mathcal{T}^X_u)$
15 **if** $\mathcal{T}^X_u$ *is finished* **then**
16 **if not** `FoundSolution`$(\mathcal{T}^X_u)$ **then**
17 **if** *u = fw* **then**
18 **return** `GenerateForwardCertificate`$(\mathcal{T}^X_u, D_{fw})$
19 **else**
20 **return** `GenerateBackwardCertificate`$(\mathcal{T}^X_u, D_{bw})$
21 **if** $X = V$ **then**
22 **return** "Solvable"
23 $SearchPool \leftarrow SearchPool \setminus \{\mathcal{T}^X_u\}$
24 $D_{\neg u} \leftarrow D_{\neg u} \cup$ `UnreachableStates`$(\mathcal{T}^X_u)$
25 `RemoveDeadEnds`$(D_{\neg u}, \mathcal{T}^X_{\neg u})$

Compared to certifying SBU certifying SymPA is only partially certifying, it does not provide a certificate if it deems a problem solvable. Though it is not extremely difficult to further extend SymPA such that it becomes fully certifying. Instead of returning "Solvable" (line 22) a plan could be extracted using the closed list of the search that proved the task solvable. Then this plan can be emitted as certificate for solvability. In this work, however, we only consider the unsolvability certificates as they are more difficult. Also SymPA is most often used for tasks that are expected to be unsolvable so unsolvability certificates are

#	judgement	rule	premises
(1)	$\emptyset$ dead	**ED**	
(2)	$D_{fw} \cap G^{\Pi} \sqsubseteq \emptyset$	**B1**	
(3)	$D_{fw} \cap G^{\Pi}$ dead	**SD**	(1), (2)
(4)	$D_{fw}[A^{\Pi}] \sqsubseteq D_{fw} \cup \emptyset$	**B2**	
(5)	D_{fw} dead	**PG**	(4), (1), (3)
(6)	$\mathcal{T}_{fw}^{X}.closed \cap G^{\Pi} \sqsubseteq \emptyset$	**B1**	
(7)	$\mathcal{T}_{fw}^{X}.closed \cap G^{\Pi}$ dead	**SD**	(1), (6)
(8)	$\mathcal{T}_{fw}^{X}.closed[A^{\Pi}] \sqsubseteq \mathcal{T}_{fw}^{X}.closed \cup D_{fw}$	**B2**	
(9)	$\mathcal{T}_{fw}^{X}.closed$ dead	**PG**	(8), (5), (7)
(10)	$\{I^{\Pi}\} \sqsubseteq \mathcal{T}_{fw}^{X}.closed$	**B1**	
(11)	$\{I^{\Pi}\}$ dead	**SD**	(9), (10)
(12)	task unsolvable	**CI**	(11)

Figure 3: SymPA unsolvability proof in forward direction

of more use than those for solvable tasks. Therefore certifying SymPA lines up with most other planners and provides a certificate only in the case it can solve more efficiently.

4.2 Unsolvability certificates

Showing unsolvability based on SymPA is essentially the same as for SBU. The proofs in both directions just integrate the dead ends found by SymPA as well. Again the goal of the proof in forward direction is to show that the initial state (or rather the set containing only the initial state) $\{I^{\Pi}\}$ is dead. This proof is used if the current search is finished (its open list is empty), it searched in forward direction and no solution was found. It does not matter whether this search $\mathcal{T}_{fw}^{X}$ is in the original search space ($X = \Pi$) or an abstract search space ($X \neq \Pi$) because if the problem is unsolvable in an abstract search space it will also be unsolvable in the original search space. A search in an abstract search space generates and expands abstract states which can be interpreted as sets of states of the original search space. So if an abstract forward search cannot reach any abstract state which contains a goal state from the abstract state containing the initial state then also in the original search space no goal can be reached from the initial state. The argument for abstract backward searches is analogous.

The following descriptions of the unsolvability proofs focus on explaining why the basic statements hold and give a high-level view of the proofs. First we will take a look at the unsolvability proof of the forward direction. For the syntactical details of this proof refer to Figure 3.

The first basic statement of the forward proof claims that the intersection between the dead ends of the forward searches D_{fw} found by the algorithm and the goal states G^{Π} is

empty (judgement 2). D_{fw} are states that the backward searches could not reach (Algorithm 2, line 24). As all backward searches start from the goal states their unreachable states are dead ends for forward searches and thus cannot contain any goal state. Hence the basic statement holds. Because the empty set is dead (judgement 1) it then directly follows that this intersection is also dead (judgement 3).

The next basic statement declares that the set of dead ends D_{fw} cannot be left by progression (judgement 4). The argument why this is true is based on that D_{fw} is the set of states which the finished backward searches discovered as definitively unreachable by regression. This holds because if an abstract state is unreachable by a backward search then all states that are represented by this abstract state must be unreachable by regression. Otherwise if one state which is represented by the abstract state were reachable by regression then the whole abstract state would be reachable by regression. Thus all other states not in D_{fw} are considered possibly reachable from the goal states by regression because they were reached by any backward search or because they are part of abstract states that were reached by any backward search.

So if there existed a state that was not included in D_{fw} but was reachable from D_{fw} by progression it would be a state that is possibly reachable by a backward search. But all states that are still considered possibly reachable by regression were reached by a backward search at some point. Then this backward search could have reached inside D_{fw} via this state and D_{fw} would not contain only states unreachable by all finished backward searches. Therefore all states that can be reached from D_{fw} are also included in this set and D_{fw} cannot be left by progression.

From combining that D_{fw} cannot be left (judgement 4) and that D_{fw} does not contain any goal state (judgement 3) it follows that D_{fw} is dead (judgement 5). Which makes sense because D_{fw} cannot be part of any plan if it neither contains a goal state nor can a goal state be reached from it.

Before using this information though we first look at the closed list of $\mathcal{T}_{fw}^X$. Similar to the dead ends D_{fw} it also does not contain any goal state (judgement 6). Because $\mathcal{T}_{fw}^X$ is finished $\mathcal{T}_{fw}^X.closed$ contains all states reachable from the initial state by progression that were not pruned. That holds because all forward searches begin with the initial state as the first search frontier and insert the successors of the current frontier into the open list as the next search frontier. The states of the old search frontier then are put into the closed list. If any goal state were put into the open list and then into the closed list this way it would be reachable from the initial state by progression because all forward searches start from the initial state. But if this had happened then the task would have been solvable and the algorithm would not have been generating the unsolvability proof. Hence no goal state was reached during this forward search and no goal state is in $\mathcal{T}_{fw}^X.closed$.

This means that the intersection of the closed list and the goal states is empty (again judgement 6). From that it directly follows that this intersection is dead (judgement 7) because the empty set is dead (judgement 1).

#	judgement	rule	premises
(1)	$\emptyset$ dead	**ED**	
(2)	$[A^\Pi]D_{bw} \sqsubseteq D_{bw} \cup \emptyset$	**B3**	
(3)	$\{I^\Pi\} \sqsubseteq \overline{D_{bw}}$	**B1**	
(4)	D_{bw} dead	**RI**	(2), (1), (3)
(5)	$[A^\Pi]\mathcal{T}^X_{bw}.closed \sqsubseteq \mathcal{T}^X_{bw}.closed \cup D_{bw}$	**B3**	
(6)	$\{I^\Pi\} \sqsubseteq \overline{\mathcal{T}^X_{bw}.closed}$	**B1**	
(7)	$\mathcal{T}^X_{bw}.closed$ dead	**RI**	(5), (4), (6)
(8)	$G^\Pi \sqsubseteq \mathcal{T}^X_{bw}.closed$	**B1**	
(9)	G^Π dead	**SD**	(7), (8)
(10)	task unsolvable	**CG**	(9)

Figure 4: SymPA unsolvability proof in backward direction

The next basic statement claims that all successor states of $\mathcal{T}^X_{fw}.closed$ are either in the closed list itself or in the set of dead ends of the forward direction D_{fw} (judgement 8). If no pruning was used then all successors of the closed list would only be in the closed list. That is because as explained earlier the closed list $\mathcal{T}^X_{fw}.closed$ contains all states reachable from the initial state by progression. However dead ends of the forward direction found by previously finished searches are removed from the open lists of all forward searches (Algorithm 2, line 25) and thus are not put into $\mathcal{T}^X_{fw}.closed$. But these pruned states are exactly the set D_{fw}. So all successor states of $\mathcal{T}^X_{fw}.closed$ are either in the closed list itself because they were reached and were moved from the open to the closed list or they are in the set of forward dead ends D_{fw} because they were pruned from the open list. Hence this basic statement holds.

It then follows that the closed list is dead (judgement 9) because all successors of $\mathcal{T}^X_{fw}.closed$ are either in $\mathcal{T}^X_{fw}.closed$ or in D_{fw} (judgement 8), D_{fw} is dead (judgement 5) and no goal state is in $\mathcal{T}^X_{fw}.closed$ (judgement 6 / 7). In other words because no goal state is in $\mathcal{T}^X_{fw}.closed$ or D_{fw} and no goal state can be reached from the closed list it cannot be part of any plan.

It is trivial that the initial state is in the closed list (judgement 10) as every forward search starts at the initial state and all visited states are put into the closed list. But with this information and the fact that the closed list is dead (judgement 9) it now follows that the initial state is dead (judgement 11). This directly leads to the conclusion of the proof. The task is unsolvable (judgement 12) because the initial state cannot be part of a plan and therefore no plan can exist at all.

Proving unsolvability for the backward direction is done by showing that the set of goal states cannot be part of a plan and is thus dead. Again in the following a high-level description of the proof is given. For the syntactical details refer to Figure 4.

The first basic statement tells that the set of dead ends of the backward direction D_{bw} cannot be left by regression (judgement 2). The argument why this holds is similar to the argument for progression in the proof of the forward direction.

D_{bw} is the set of discovered dead ends for backward searches and this is exactly the set of states that were found to be unreachable by the finished forward searches. All states not in D_{bw} are still considered possibly reachable by a forward search because they were reached by a forward search or because they are part of an abstract state that was reached by a forward search. If there was a state not included in D_{bw} but reachable from D_{bw} by regression it would be a state considered possibly reachable by a forward search. Hence it would have been reached by a forward search at some point and then this forward search could have reached inside D_{bw} with progression via this state. But then D_{bw} would not anymore include only states unreachable by progression. Therefore no state can exist that is reachable from D_{bw} by regression and not in D_{bw} so the basic statement holds.

For the first deduction a second basic statement is needed. It says that the initial state is not part of D_{bw} (judgement 3). Considering the argumentation for the last basic statement this is true because D_{bw} includes states not reachable by any forward search but the initial state is the starting point of all forward searches. So it will never be part of D_{bw} and thus the basic statement holds.

Using these two basic statement it can be concluded that D_{bw} is dead (judgement 4). D_{bw} cannot be left by regression and the initial state is not part of this set hence D_{bw} can never reach the initial state and so it cannot be part of any plan.

The next basic statement claims that the predecessors of the closed list of the current search $\mathcal{T}_{bw}^{X}$ are either in the closed list $\mathcal{T}_{bw}^{X}.closed$ itself or in the set of dead ends D_{bw} (judgement 5). Without pruning all predecessors of $\mathcal{T}_{bw}^{X}.closed$ would be in this set only. That is the case because starting from the set of goal states as the first search frontier all backward searches move the current search frontier from the open list to the closed list and put its predecessors in the open list as the new frontier during one search step. This means that all states reachable by regression would be in $\mathcal{T}_{bw}^{X}.closed$. However D_{bw} is used to prune all states that were detected to be dead ends for backward searches. So all states removed this way are in D_{bw} and hence the predecessors of $\mathcal{T}_{bw}^{X}.closed$ are either in the closed list itself if they were not detected to be dead ends or they are in D_{bw} if they were detected as dead ends.

Before deriving the next step another basic statement is needed. The initial state is not included in $\mathcal{T}_{bw}^{X}.closed$ (judgement 6). If it were in the closed list of the current backward search then the task would be solvable because the closed list contains all states that are reachable from the goal states by the current backward search. But if the task were solvable then the algorithm would not generate an unsolvability proof. Therefore the initial state cannot be in $\mathcal{T}_{bw}^{X}.closed$.

Knowing that D_{bw} is dead, that the initial state is not in $\mathcal{T}_{bw}^{X}.closed$ and that all predecessors of this closed list are either in itself or in D_{bw} it can be concluded that $\mathcal{T}_{bw}^{X}.closed$ is dead (judgement 7).

With this information the last basic statement needed to finish the proof is that the set of all goal states is part of $\mathcal{T}_{bw}^{X}.closed$ (judgement 8). This holds because all backward searches have the set of all goal states as their starting points. So the goal states are moved from the open list of a backward search to the closed list during a search step.

It now follows that the set of all goal states is dead (judgement 9) because it is part of $\mathcal{T}_{bw}^{X}.closed$ which is dead itself. Therefore to finish the proof this information is used to conclude that the task is unsolvable (judgement 10) because no goal state can be part of a plan.

5 Implementation

The extension of SymPA was fully implemented in C++ on top of SymPA. SymPA itself is build on the Fast Downward Planning System (Helmert, 2006). Because of this SymPA and also the added generation of unsolvability certificates can handle tasks written in PDDL (Ghallab et al., 1998) as well as SAS$^+$ (Bäckström and Nebel, 1995).

To integrate certificate generation in SymPA no big changes had to be made. Certificate generation is called after the search finished and no solution was found right before the algorithm terminates. In the case of SymPA one advantage of the used proof system is that all needed information is available when the planner finished its search. Nothing of the actual search had to be altered which also means that the runtime of the search is not influenced by certificate generation. The overall runtime however does increase for unsolvable problems because certificate generation is appended.

One important aspect of extending SymPA was to make sure that the generated certificates are compatible with the verifier that checks the certificates. Three files have to be created for an unsolvability certificate when using the verifier of Eriksson (2019b). The first file stores the BDDs which are used in the proof. In our case it stores the BDD of the closed list and if dead ends were found they are stored as a second BDD in this file. Next a text file is needed which holds a description of the planning task that was given to SymPA. The original task description cannot be used here because the task file needs to be in a specific format such that the verifier can process it. Finally a text file with the actual proof has to be created. This file lists the judgements of Proof 3 or 4 in a format that the verifier can process.

One difficulty of generating these components was that BDDs store binary variables but in the original task the domains of the variables could include more than two values. To overcome this a conversion of the original variables to binary variables is made. Every variable v with domain D_v is represented in the BDDs with $\lceil \log_2 |D_v| \rceil$ binary variables

which can encode all values from D_v. This conversion is reflected in the task file. The description of the initial state lists all binary variables that have to be true in the initial state. The goal description is split into positive and negative goals where the former lists all binary variables that have to be true and the later lists all binary variables that have to be false in any goal state. The description of preconditions and effects of the actions work accordingly.

Furthermore the BDDs in SymPA store for every binary variable a second binary variable which is needed for the computation of state expansions. These so called primed variables are not part of the actual task and are not needed for certificate generation but they cannot be ignored because they still exist in the BDDs so the verifier sees them as part of the task. One way to handle this could be to modify the BDDs and remove the primed variables. The method we chose however alters the task description for the verifier. Instead of listing only the binary variables needed, the task description lists the primed variables interleaved with the needed binary variables. This causes no problems because the primed variables are not actually used in the BDDs or the task description. But this allows us to use SymPAs BDDs without modification.

6 Experiments

For the experimental evaluation of certifying SymPA we used the same planning tasks as were used in the Unsolvability IPC 2016[1]. The experiments were run on a cluster consisting of Xeon E5-2660 (2.2 GHz) processors. A total time limit of 30 minutes and a total memory limit of 3584MiB was used. Additionally 2GB were set as memory limit for each translation and search. For SymPA the maximal number of nodes a search frontier may include was set to 10000 and mutexes were disabled. The Downward Lab toolkit (Seipp et al., 2017) was used as framework for running the experiments.

For the 352 tasks that were examined SymPA successfully finished its search and the certificate generation in 105 cases where it correctly deemed all tasks as unsolvable. Of these 105 runs where unsolvability certificates were generated 79% (83) of the certificates were valid. The average size of a certificate including all three files (task description, BDDs and proof) is 5560KB and the largest certificate was in the *over-nomystery-uns16* domain with 101567KB. The geometric mean of the time the planner needed is 3.29 seconds of which 0.40 seconds were needed for the search. The geometric mean of the time the verifier needed is 45.32 seconds and the average of the total memory needed by all runs is 372MiB.

247 runs did not finish the search and thus were not verified for various reasons. 70% (174) of these runs terminated due to timeouts and another 10% (25) because they went out of memory. The remaining 20% (48) of the runs terminated because of other kinds of

[1]https://unsolve-ipc.eng.unimelb.edu.au/ (last accessed 05.09.2020)

errors. All runs of the *over-tpp-uns16* domain which contains 30 tasks terminated because of a problem with the domain definition. The remaining 18 runs that terminated because of an error are mostly from the *bag-gripper-uns16* domain and are also related to memory problems.

For the 22 certificates that could not be verified one of the following three judgements does not hold:

Proof 3, judgement 2: $D_{fw} \cap G^{\Pi} \sqsubseteq \emptyset$

Proof 3, judgement 4: $D_{fw}[A^{\Pi}] \sqsubseteq D_{fw} \cup \emptyset$

Proof 4, judgement 2: $[A^{\Pi}]D_{bw} \sqsubseteq D_{bw} \cup \emptyset$

All three judgements make a statement about the set of dead ends. This leads to the assumption that the same reason could prevent the certificates from being valid. At least for the two later judgements it probably is one source that causes them not to be true.

First we look at judgement 2 from Proof 3. If this judgement does not hold then an abstract backward search must not have been able to reach at least one of the goal states. This is the only possibility for D_{fw} to contain a goal state and thus for the intersection between D_{fw} and G^{Π} to be empty. It must be an abstract backward search because only abstract searches can expand the dead ends. Furthermore forward searches cannot modify the dead ends of the forward direction because the dead ends of one direction are always discovered by a search of the other direction (Algorithm 2, line 24). We could already detect inside SymPA that this intersection is not empty. Thus we can rule out a faulty task description for this issue.

The cause of this probably is connected with the initialization of new abstract searches. The backward search in the original search space is initialized with the goal states and thus after the first expansion its closed list contains the goal states. Before a new abstract search however inherits the open and closed list from the last search of the same direction they are relaxed. This relaxation might lead to forgetting some of the goal states in the new closed list and if these forgotten goal states are unreachable for the new abstract search then they are put into the set of dead ends of the forward direction. These forgotten goal states are unreachable by regression for this particular abstract search but as they are goal states this is not relevant. They have to be reachable by progression which might still be possible but cannot be detected by this abstract backward search. A forward search could in theory reach into D_{fw} with progression, thus if the forgotten goal states are in D_{fw} this search might be able to reach one of these goal states. This possibility is not discovered though because all states in D_{fw} are pruned.

Hence a possible plan could be overlooked. This is no problem however for the experiments that were run because no solvable tasks were part of the benchmarks. A solution for this issue could be to manually remove all goal states from D_{fw} after all states unreachable by regression were put into it.

The following two judgements are counterparts and presumably have the same cause that leads them to not being true. Judgement 4 of Proof 3 states that the set of discovered dead ends of the forward direction cannot be left by progression. Judgement 2 of Proof 4 accordingly describes that the set of discovered dead ends for the backward direction cannot be left by regression.

We suspect that the reason that these two judgements do not hold is related to how the unreachable states are computed. When an abstract search finishes, the union of its own closed list is build with the closed lists of all searches in the same direction u that started before the current search. The negation of this union is returned as unreachable states in direction u. These previous searches are not all finished though. At least the search in the original search space in the same direction cannot be finished. Otherwise it would have found a plan or found out that no plan exists and terminated the algorithm. So at least one closed list of a not finished search is used to find all reachable states. But this closed list might not contain all reachable states and if none of the following abstract searches in this direction reaches those missing states then they are in no closed list of direction u. Therefore they must be in the negation of the union of all closed lists of this direction. This negation however is used as set of unreachable states in direction u and furthermore as $D_{\neg u}$, the set of dead ends for the other direction. This means that $D_{\neg u}$ might contain states that are reachable in direction u but then these states are no dead ends in direction $\neg u$. Hence D_{fw} could possibly be left by progression and D_{bw} could possibly be left by regression which means that the above judgements indeed do not hold in general.

Similar to the previously discussed issue these results cause no further problems for the experiments. Again states that might lead to a plan are pruned. So potentially solvable tasks might be declared unsolvable which could not have happened in our experiments because no solvable tasks were used. One option to fix this could be to only use the states unreachable by the current abstract search. With this perhaps less dead ends are pruned which could impact performance but no solvable tasks would be overlooked.

7 Related Work

In the context of correctness guarantees model checking can be seen as a complement to planning. Model checking tries to verify correctness of a hardware or software system in the sense that a system is correct if from the initial state no "erroneous" state can be reached. Take for example an elevator which should never have opened doors and be moving at the same time. To ensure the passengers safety this error state must never be reached. So compared to planning where we try to find a path from the initial state to a goal state model checking tries to show that no path from the initial state to any error state exists.

Even similar methods to handle the state explosion problem which is present in both areas can be used. For symbolic search which we discussed earlier there exists symbolic

model checking as an equivalent. Similar to SymPA symbolic model checking as presented by Burch et al. (1992) also uses BDDs to represent the state space. An approach even more similar to SymPA is presented by Coudert et al. (1989) which uses BDDs for symbolic breadth first traversal of states of a machine.

8 Future Work

Our version of SymPA gives correctness guarantees for its result if it detects a task to be unsolvable. There is still room for improvement however. For example SymPA could be extended further to a fully certifying algorithm which generates plans for solvable tasks and the presented certificates for unsolvable tasks.

Besides SymPA the unsolvability proof system could be applied to other planning algorithms to make them partially or even fully certifying. The theoretical proofs for SBU which we covered in Section 3.2 could be used to implement an extension for SBU to make it a fully certifying algorithm. With that in mind also SymBA* (Torralba, 2015) could be extended to a fully certifying algorithm. Considering the similarities between SymBA* and SymPA and especially between SymBA* and SBU the theoretical unsolvability proofs for SymBA* should not be quite difficult to obtain.

9 Conclusion

Certifying SymPA combines SymPA's ability to efficiently detect tasks as unsolvable with certificate generation to verify the planners results. In addition to the theory behind certifying SymPA we presented an implementation of certifying SymPA, its experimental evaluation and unsolvability proofs for a possible extension of symbolic bidirectional uniform-cost search.

Certifying algorithms provide a quite feasible option to give correctness guarantees at least for specific inputs. Most planning algorithms emit a plan for solvable tasks and are thus already partially certifying. Applying the unsolvability proof system to these algorithms can enhance them to give correctness guarantees for all valid planning tasks and not only solvable ones. It is more impactful however if the proof system is applied to algorithms that are not certifying at all. As can be seen with certifying SymPA this enables these algorithms to give at least some correctness guarantees which is a big difference to giving no correctness guarantees at all.

References

Bäckström, C. and Nebel, B. (1995). Complexity results for SAS^+ planning. *Computational Intelligence*, 11(4):625–655.

Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L.-J. (1992). Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170.

Coudert, O., Berthet, C., and Madre, J. C. (1989). Verification of synchronous sequential machines based on symbolic execution. In *International Conference on Computer Aided Verification*, pages 365–373. Springer.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

Eriksson, S. (2019a). *Certifying planning systems: witnesses for unsolvability*. PhD thesis, University of Basel.

Eriksson, S. (2019b). Code from the PhD thesis "Certifying planning systems: witnesses for unsolvability". `https://doi.org/10.5281/zenodo.3355459`.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208.

Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). *PDDL—The Planning Domain Definition Language Version 1.2*. Yale Center for Computational Vision and Control, Technical Report CVC TR-98-003/DCS TR-1165.

Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246.

Helmert, M. (2009). Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535.

McConnell, R. M., Mehlhorn, K., Näher, S., and Schweitzer, P. (2011). Certifying algorithms. *Computer Science Review*, 5(2):119–161.

Seipp, J., Pommerening, F., Sievers, S., and Helmert, M. (2017). Downward lab. `https://doi.org/10.5281/zenodo.399255`.

Torralba, Á. (2015). *Symbolic Search and Abstraction Heuristics for Cost-Optimal Planning*. PhD thesis, Universidad Carlos III de Madrid.

Torralba, Á. (2016). SymPA: Symbolic perimeter abstractions for proving unsolvability. *Unsolvability International Planning Competition 2016 planner abstracts*, pages 8–11.