Journal of Artificial Intelligence Research 50 (2014) 265-319 Submitted 10/13; published 06/14

Property Directed Reachability for Automated Planning

Martin Suda SUDA@MPI-INF.MPG.DE
Max-Planck-Institut für Informatik,
Saarbrücken, Germany
Charles University, Prague, Czech Republic

Abstract

Property Directed Reachability (PDR) is a very promising recent method for deciding reachability in symbolically represented transition systems. While originally conceived as a model checking algorithm for hardware circuits, it has already been successfully applied in several other areas. This paper is the first investigation of PDR from the perspective of automated planning.

Similarly to the planning as satisfiability paradigm, PDR draws its strength from internally employing an efficient SAT-solver. We show that most standard encoding schemes of planning into SAT can be directly used to turn PDR into a planning algorithm. As a non-obvious alternative, we propose to replace the SAT-solver inside PDR by a planning-specific procedure implementing the same interface. This SAT-solver free variant is not only more efficient, but offers additional insights and opportunities for further improvements. An experimental comparison to the state of the art planners finds it highly competitive, solving most problems on several domains.

1. Introduction

Property Directed Reachability (PDR), also known as IC3, is a recently proposed algorithm for deciding reachability in symbolically represented transition systems (Bradley, 2011; Eén, Mishchenko, & Brayton, 2011).[1] Since its discovery in 2010, it has already established itself as one of the strongest model checking algorithms used in hardware verification. The original and inspiring way in which PDR harnesses the power of a modern SAT-solver gives the algorithm a unique ability to discover long counterexample paths combined with a remarkable performance in proving unreachability. Other interesting traits include a typically small memory footprint and a good potential for parallelization.

With awareness of the well-known equivalence between model checking and automated planning, the aim of this work is to investigate PDR from the planning perspective. Our main goal is to establish whether the practical success of the algorithm can be repeated on planning benchmarks. Moreover, we are also interested in the relation of PDR to the currently used planning techniques. It is illustrative for highlighting some of the interesting features of the algorithm to start with a preliminary comparison right away.

The fact that PDR builds upon the SAT-solving technology makes it obviously related to the *planning as satisfiability* approach (Kautz & Selman, 1996). While in planning as satisfiability the underlying SAT-solver receives formulas of increasing size as the method progresses to check for existence of increasingly longer plans, in PDR each SAT-solver call

1. IC3 is the name Aaron Bradley, the originator of the algorithm, gave to the first implementation (Bradley, 2011). The more descriptive name Property Directed Reachability was coined by Eén et al. (2011).

corresponds only to a single step in the transition system. That is why the algorithm is sometimes said to operate "without unrolling". Dealing with formulas over a fixed signature lifts some of the computational burden from the SAT-solver, making it respond in a more reliable way.

Similarly to planning as satisfiability, PDR proceeds iteratively, gradually disproving existence of plans of length 0, 1, 2,. . . . Due to a so-called obligation rescheduling technique, however, PDR can discover a plan of length l already during iteration $k < l$, that is, while the existence of shorter plans has not necessarily been ruled out yet. This typically leads to improved performance, as it allows the algorithm to avoid completing the potentially expensive non-existence proofs. A similar effect can be achieved in the planning as satisfiability approach by running several SAT-solvers in parallel or interleaved (Rintanen, 2004). Such a modification, however, requires a non-trivial engineering effort and the resulting system contains parameters that need to be tuned for the problem at hand. In contrast, rescheduling in PDR amounts literally to a one-line change in the algorithm. When this line is disabled, PDR resorts to the more expensive search for an optimal length plan.

Surprisingly, PDR can also be naturally compared to the explicit *heuristic search planning* (Bonet & Geffner, 2001). Indeed, PDR is probably best understood as a hybrid between explicit and symbolic approaches. Because the satisfying assignment in PDR is built systematically, one transition at a time, the finished part of it corresponds to an explicit path through the transition system and its last piece to a state to be expanded next. At the same time, PDR maintains symbolic reachability information in a form of a sequence of sets of clauses. The k-th set in the sequence over-approximates the k-fold preimage of the set of goal states. These clause sets play a role similar to an admissible heuristic. They represent a lower bound estimate for the distance of a state to the goal and thus provide a means to guide the search towards it. However, while a heuristic value of a particular state is normally computed only once and it remains constant during the search for a plan, the clause sets in PDR are refined continually. The refinement happens on demand, driven by the states encountered during the search.

1.1 Paper Overview

In order to apply PDR to a planning problem, the problem has to be processed into a suitable form. In Section 2 we introduce a symbolic transition system, a description of a reachability task based on the clausal language of propositional logic, which serves as a generic input for PDR. We observe that most of the standard encoding schemes of planning into SAT provide us with such a description. This means that a general implementation of the algorithm, which we describe in detail in Section 3, combined with any such encoding already yields a stand-alone planner. This is, however, not the most efficient path to take.

The main contribution of this paper is presented in Section 4. We show that instead of relying on an encoding and a general purpose SAT-solver, we can, at least in the case of the sequential plan semantics, delegate the single step reachability queries to a planning-specific procedure. Not only do we gain a polynomial time guarantee for answering the individual queries, but by decoupling PDR from the underlying SAT-solver we also gain additional insights and ideas for further improvements.

We implemented the proposed idea in a new planner PDRplan. In Section 5 we experimentally confirm that it is more efficient than the standard PDR combined with encodings. We also evaluate the practical impact of various improvements, and compare the most successful configuration of PDRplan to the state of the art planners with encouraging results.

Section 6 returns to related work and uncovers a perhaps surprising connection between PDR and the Graphplan algorithm of Blum and Furst (1997). Finally, Section 7 uses examples of the behavior of PDR on two classical planning domains to discuss possibilities for future extensions of the algorithm and Section 8 concludes.

2. Preliminaries

In this section we build the necessary background for explaining PDR. After recalling the basic notions of propositional logic and fixing the notation, we introduce symbolic transition systems, which serve as a canonical input for the algorithm. We observe that the encoding part of the well-known planning as satisfiability paradigm can be seen to translate a given planning problem into a symbolic transition system. This means that by combining an encoding with PDR one already obtains a standalone planner.

2.1 Propositional Logic

A *signature* Σ is a finite set of propositional *variables*. Formulas are built from variables using the propositional connectives *negation* $\neg$, *conjunction* $\wedge$, *disjunction* $\vee$, and *implication* $\Rightarrow$. Given a formula F over Σ we denote by $\mathit{Vars}(F)$ the set of variables actually occurring in F (i.e. $\mathit{Vars}(F) \subseteq \Sigma$). A *literal* l is either a variable $p \in \Sigma$ or its negation $\neg p$. In the first case, the literal is called *positive*. We define the *complement* $\neg l$ of a literal l to be $\neg p$ if $l = p$, and to be p if $l = \neg p$. Also in other contexts, double negations are silently reduced away.

A consistent conjunction of literals is referred to as a *cube* and a disjunction as a *clause*. A cube r is *full* if $\mathit{Vars}(r) = \Sigma$. When describing algorithms, it is advantageous to treat both cubes and clauses simply as sets of literals and leave the interpretation to follow from the context. Then, by *complementing* a cube r we obtain a clause $\neg r = \{\neg l \mid l \in r\}$ and also vice versa. We call a clause *positive* if all its literals are positive. A clause c is said to *subsume* another clause d if $c \subseteq d$. As usual, sets of clauses stand for their conjunction.

The semantics of propositional logic is built around the notion of an *assignment*, which is a mapping $s : \Sigma \to \{\mathbf{0}, \mathbf{1}\}$ from the signature Σ to the truth values $\{\mathbf{0}, \mathbf{1}\}$. We write $s \models F$ if an assignment s satisfies a formula F. A formula F is called *satisfiable* if there is an assignment that satisfies it, and it is called *valid* if $\neg F$ is not satisfiable. Assignments naturally correspond to full cubes: Given an assignment s we define a full cube

$$\mathit{Lits}(s) = \{p \mid p \in \Sigma \text{ and } s(p) = \mathbf{1}\} \cup \{\neg p \mid p \in \Sigma \text{ and } s(p) = \mathbf{0}\}.$$

There is exactly one assignment, namely s, which satisfies $\mathit{Lits}(s)$.

2.2 Encoding Discrete Time

When reasoning about systems which evolve in time, we use the basic signature $\Sigma = \{p, q, \ldots\}$ to describe the current state of the system and introduce a disjoint copy of Σ

denoted $\Sigma' = \{p', q', \dots\}$ to represent the state of the system after one step. Similarly, further copies $\Sigma'', \Sigma''', \dots$ (also written as $\Sigma^{(2)}, \Sigma^{(3)}, \dots$) stand for the states further in the future. The priming notation is extended to formulas and assignments in the following way. By F' we denote the formula obtained from a formula F by priming every variable occurring in F. For an assignment $s : \Sigma \to \{\mathbf{0}, \mathbf{1}\}$, we denote by $s' : \Sigma' \to \{\mathbf{0}, \mathbf{1}\}$ the assignment that behaves on primed symbols as s does on unprimed ones, i.e., $s'(p') = s(p)$ for every $p \in \Sigma$. If s and t are two assignments from Σ, we let (s, t') denote the joint assignment $(s \cup t') : \Sigma \cup \Sigma' \to \{\mathbf{0}, \mathbf{1}\}$. This means that

$$(s, t')(x) = \begin{cases} s(p) & \text{if } x = p \ \in \Sigma, \\ t(p) & \text{if } x = p' \in \Sigma'. \end{cases}$$

Such an assignment gives a truth value to formulas over the joint signature $\Sigma \cup \Sigma'$.

2.3 Symbolic Transition Systems

A *symbolic transition system* (STS) is a tuple $\mathcal{S} = (\Sigma, I, G, T)$, where Σ is a signature, I, called the *initial* formula, and G, the *goal* formula, are sets of clauses over Σ, and T, the *transition* formula, is a set of clauses over $\Sigma \cup \Sigma'$. An STS $\mathcal{S}$ symbolically represents an explicit transition system $\mathcal{T}_\mathcal{S} = (S, S_I, S_G, R_T)$, which we describe next. Notice that the symbolic representation can be exponentially more succinct than the explicit system. The explicit transition system $\mathcal{T}_\mathcal{S}$ consists of

- the set of states S, identified with the set of all assignments from Σ:

$$S = \{s \mid s : \Sigma \to \{\mathbf{0}, \mathbf{1}\}\},$$

- a subset $S_I \subseteq S$ of the initial states, which are states that satisfy the initial formula:

$$S_I = \{s \in S \mid s \models I\},$$

- a subset $S_G \subseteq S$ of the goal states, which are states that satisfy the goal formula:

$$S_G = \{s \in S \mid s \models G\},$$

- and the transition relation $R_T \subseteq S \times S$ of pairs of states (also called transitions) which jointly satisfy the transition formula:

$$R_T = \{(s, t) \mid s, t \in S \text{ and } (s, t') \models T\}.$$

A *path* in $\mathcal{T}_\mathcal{S}$ is a finite sequence $s_0, \dots, s_k$ of states such that $(s_j, s_{j+1}) \in R_T$ for every $j = 0, \dots k - 1$. We will be interested in the existence of paths connecting an initial state with a goal state. We say that an STS $\mathcal{S}$ is *satisfiable* if there is a path $s_0, \dots, s_k$ in $\mathcal{T}_\mathcal{S}$ such that $s_0 \in S_I$ and $s_k \in S_G$. For simplicity, we call such a path a *witnessing* path for $\mathcal{S}$.

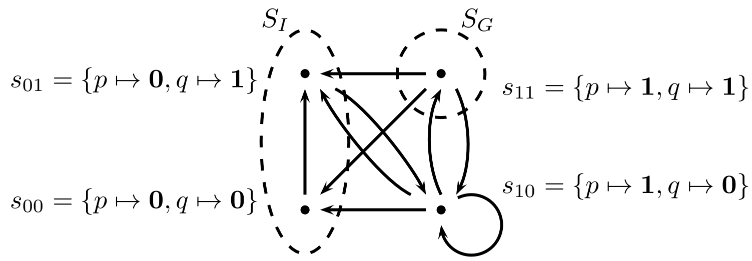

Figure 1: The explicit transition system $\mathcal{T}_\mathcal{S}$ represented by the STS $\mathcal{S}$ from Example 1. Its four states correspond to the four assignments over the signature $\Sigma = \{p, q\}$.

Example 1. Consider an STS $\mathcal{S} = (\Sigma, I, G, T)$, where $\Sigma = \{p, q\}$, $I = \{\neg p\}$, $G = \{p, q\}$, and

$$T = \{p \vee \neg p' \vee \neg q',\ p \vee q \vee \neg p',\ p \vee q \vee q',\ p \vee \neg q \vee p',\ \neg q \vee \neg p' \vee \neg q'\}.$$

Notice that we prefer the formula notation (as opposed to the set notation) for concrete clauses. This means that I consist of one and G of two unit clauses, i.e. clauses with just a single literal. The corresponding explicit transition system $\mathcal{T}_\mathcal{S}$ is shown in Figure 1. The path $s_{00}, s_{01}, s_{10}, s_{11}$ is an example of a witnessing path for $\mathcal{S}$. The STS $\mathcal{S}$ is satisfiable.

It is useful to notice that the definition of an STS is symmetrical in the following sense. Given an STS $\mathcal{S} = (\Sigma, I, G, T)$ an *inverted* STS is defined as $\mathcal{S}^{-1} = (\Sigma, G, I, T^{-1})$, where T^{-1} is obtained from T by simultaneously removing primes from all the occurrences of primed variables and adding primes to all the occurrences of originally unprimed variables. This corresponds, on the explicit side, to exchanging the initial and goal states and inverting the direction of all the transitions. Therefore, an STS $\mathcal{S}$ is satisfiable if and only $\mathcal{S}^{-1}$ is. Moreover, a witnessing path for $\mathcal{S}$ can be recovered from a witnessing path for $\mathcal{S}^{-1}$ (also vice versa) by reading the respective sequence backwards.

2.4 Propositional STRIPS Planning

In this paper we work with planning problems described in the STRIPS planning formalism. Similarly to states in transition systems, states of the world in STRIPS planning are identified with propositional assignments. The propositional variables encoding the state are in this context called *state variables* and we denote their set by X.

An action a is determined by a tuple $a = (\mathit{pre}_a, \mathit{eff}_a)$, where pre_a, called the *precondition list*, and eff_a, the *effect list*, are cubes over X, i.e. consistent conjunctive sets of literals. An action a is *applicable* in a state s if $s \models \mathit{pre}_a$. If this is the case then applying the action a in s results in a *successor* state $t = \mathit{apply}(s, a)$, which is the unique state that satisfies eff_a and for every $p \in X$ not occurring in eff_a it has $t(p) = s(p)$. A degenerate action with empty precondition and effect lists is called the *noop* action. It is applicable in any state s and the corresponding successor is identical to the original state: $\mathit{apply}(s, \mathit{noop}) = s$.

A *STRIPS planning problem* is a tuple $\mathcal{P} = (X, s_I, g, \mathcal{A})$, where X is the set of state variables, s_I the initial state, g the goal condition in the form of a cube over X, and $\mathcal{A}$ a set of actions. A *plan* for $\mathcal{P}$ is a finite sequence $a_1, \ldots, a_k$ of actions from $\mathcal{A}$ such that there are states $s_0, \ldots, s_k$ satisfying the following conditions:

- $s_0 = s_I$,
- a_j is applicable in s_{j-1} for $j = 1, \dots, k$,
- $s_j = apply(s_{j-1}, a_j)$ for $j = 1, \dots, k$,
- and $s_k \models g$.

Notice that the empty sequence λ is a plan for $\mathcal{P}$ if and only if $s_I \models g$.

2.5 Planning as Satisfiability

The basic idea behind the planning as satisfiability paradigm (Kautz & Selman, 1992, 1996) is as follows. Given a planning problem $\mathcal{P}$ we define a sequence of propositional formulas $F_0, F_1, \dots$ such that there is a plan for $\mathcal{P}$ if and only if a formula F_i is satisfiable for some i. The individual formulas F_i are then iteratively checked using a SAT-solver and when a satisfiable F_i is found, a plan is recovered from the corresponding satisfying assignment.

The concrete form of the formulas in the sequence is dictated by an *encoding scheme* (see, e.g., Kautz, McAllester, & Selman, 1996; Rintanen, Heljanko, & Niemelä, 2006; Huang, Chen, & Zhang, 2012). Most encoding schemes have a simple structure that can be captured by an STS $\mathcal{S} = (\Sigma, I, G, T)$, from which the individual formulas F_i are then obtained as

$$F_i = I \wedge T^{(0)} \wedge T^{(1)} \wedge \ldots \wedge T^{(i-1)} \wedge G^{(i)}. \tag{1}$$

Note that we use the priming notation as described in Section 2.2 and thus $T^{(0)}$ stands for the same formula as T, $T^{(1)}$ for the same formula as T', etc. The resulting formula (1) for F_i, which is over signature $\bigcup_{j=0,\dots,i} \Sigma^{(i)}$, expresses the existence of a witnessing path for $\mathcal{S}$ of length i. If the encoding scheme uses a so called *sequential plan* semantics, such a witnessing path also directly corresponds to a plan of length i. This is equivalent to saying that the transition relation encoded by T allows for application of a single action in one step. *Parallel plan* semantics allow multiple actions to be applied in one time step. This leads to a more compact representation and potentially faster discovery of plans. Additional conditions on the parallel actions need to be imposed, however, to guarantee that a true sequential plan can be recovered in the end (see Rintanen et al., 2006, for more details).

2.6 Two Simple Encodings

We close this section by introducing two example encodings of a STRIPS planning problem $\mathcal{P} = (X, s_I, g, \mathcal{A})$ into an STS. They are perhaps the simplest representatives of encoding schemes with the sequential and parallel plan semantics, respectively. We will later refer to them in our theoretical considerations.

The transition systems $\mathcal{S}^{seq}_{\mathcal{P}}$ and $\mathcal{S}^{par}_{\mathcal{P}}$ corresponding to the two encodings share several building blocks. Let the signature Σ consist of the state variables X in union with a set of fresh auxiliary variables $A = \{p_a \mid a \in \mathcal{A}\}$ used for encoding applied actions. Further, let us identify the initial formula I with the cube $Lits(s_I)$ and define the goal formula G by reinterpreting the goal condition g, which is formally a cube, as a set of unit clauses $G = \{\{l\} \mid l \in g\}$. The action mechanics is in both encodings captured by the following *action precondition* axioms AP and *action effect* axioms AE:

$$AP = \{\neg p_a \vee l \mid a \in \mathcal{A},\ l \in pre_a\}, \qquad AE = \{\neg p_a \vee l' \mid a \in \mathcal{A},\ l \in \mathit{eff}_a\}.$$

The encodings differ in how they formalize the "preserving" part of actions' semantics.

The *sequential encoding* $\mathcal{S}_{\mathcal{P}}^{seq}$ relies on the so called *classical frame* axioms CF (McCarthy & Hayes, 1969) complemented by the single *at-least-one* axiom $alo = \bigvee_{a \in \mathcal{A}} p_a$:

$$CF = \{\neg p_a \vee l \vee \neg l' \mid a \in \mathcal{A},\ l \text{ literal over } X \text{ such that } l \notin \mathit{eff}_a \text{ and } \neg l \notin \mathit{eff}_a\}.$$

Putting these together, we obtain $\mathcal{S}_{\mathcal{P}}^{seq} = (\Sigma, I, G, T^{seq})$, where $T^{seq} = AP \wedge AE \wedge CF \wedge alo$. Note that the at-least-one axiom is needed, because without it a transition into an arbitrary state would be possible from a state where no action is applied, i.e. a state in which p_a is false for every $a \in \mathcal{A}$. On the other hand, the classical frame axioms ensure that if two actions are applied together in a state their effects must be identical. Thus when extracting a (sequential) plan from a witnessing path for $\mathcal{S}_{\mathcal{P}}^{seq}$ we can arbitrarily choose in each step any action $a \in \mathcal{A}$ such that p_a is true in the corresponding state.

The *parallel encoding* $\mathcal{S}_{\mathcal{P}}^{par}$ uses the following *explanatory frame* axioms EF (Haas, 1987)

$$EF = \{l \vee \neg l' \vee \textstyle\bigvee_{a \in \mathcal{A}\ l \in \mathit{eff}_a} p_a \mid l \text{ literal over } X\},$$

in combination with the so called *conflict exclusion* axioms CE

$$CE = \{\neg p_a \vee \neg p_b \mid a, b \in \mathcal{A},\ a \neq b,\ \text{and the actions } a \text{ and } b \text{ are conflicting}\},$$

where two actions are considered *conflicting* if one's precondition is inconsistent with the other's effect, i.e. if there is a literal l over X such that

$$\text{either } l \in \mathit{pre}_a \text{ and } \neg l \in \mathit{eff}_b, \text{ or } l \in \mathit{pre}_b \text{ and } \neg l \in \mathit{eff}_a.$$

In sum, we define $\mathcal{S}_{\mathcal{P}}^{par} = (\Sigma, I, G, T^{par})$ where $T^{par} = AP \wedge AE \wedge EF \wedge CE$. In this encoding two actions can be applied in parallel if they have consistent effects (action effect axioms) and one does not destroy a precondition of the other (conflict exclusion axioms). When recovering a sequential plan, such parallel actions can be serialized in any order.

Please consult the work of Ghallab, Nau, and Traverso (2004, ch. 7.4) for further details.

3. Property Directed Reachability

In this section we present PDR as an algorithm for deciding satisfiability of symbolic transition systems. The algorithm is probably best understood as an *explicit* search through the given transition system complemented by *symbolic* reachability analysis. On the explicit side, it constructs a path starting from an initial state and extending it step by step towards the goal.[2] At the same time, it maintains symbolic stepwise approximating reachability information, which is locally refined whenever the current path cannot be extended further. The reachability information guides the path construction and is also bound to eventually converge to a certificate of non-reachability, if no witnessing path exists.

2. In the standard formulation, PDR actually builds the path the other way round, from a goal state backwards towards an initial state. This is only a small detail from the theory point of view, since the definition of an STS is symmetrical. On the other hand, as we later show, the direction we adopt here gives rise to a much more successful algorithm on typical planning benchmarks.

3.1 Extension Query and Reason Computation

Let us assume an STS $\mathcal{S} = (\Sigma, I, G, T)$ is given. While the basic building block for the constructed path is a state, the reachability information is composed of sets of clauses. The core operation, around which the algorithm is built, is *extending* the current path by one step. Given a state s and a set of clauses L, we ask whether there is a state t, a successor of s with respect to T, satisfying the clauses of L. Such a question can be delegated to a SAT-solver by posing the following query:

$$SAT?[\,Lits(s) \wedge T \wedge (L)'\,]. \tag{2}$$

If the answer is positive, we can extract a successor state t from the satisfying assignment, which is necessarily of the form (s, t'), and extend the current path. In the unsatisfiable case, we compute a reason for why s does not have a successor with the property L. A *reason* is a cube $r \subseteq Lits(s)$ such that already the formula $r \wedge T \wedge (L)'$ is unsatisfiable. PDR then removes s from the path and learns the clause $c = \neg r$ to prevent the same situation from happening in the future. The clause c is a property of the preimage of L with respect to T which the state s fails to satisfy.

It is important for the efficiency of the algorithm to always compute a reason which is as small as possible. This is because a reason with fewer literals gives rise to a shorter clause, which better generalizes from the current situation. After such a clause is learned not only the state s but also many similar states will be known to have no successor satisfying L.

There are several techniques for computing small reasons. Below we describe two of them: SAT-solving under assumptions and explicit minimization. We postpone discussing a third one, inductive minimization, till Section 3.5. It is important for the correctness of PDR that the final reason is not consistent with the goal formula G. We close this section by explaining how this property can be achieved.

3.1.1 SAT-solving under Assumptions

Many modern SAT-solvers do not simply return UNSAT, but are able to identify which of the input clauses were actually used in the derivation of unsatisfiability. Solving under assumptions is a particular, simple and efficient form of such unsatisfiable core extraction technique, first introduced by Eén and Sörensson (2003) in their SAT-solver Minisat. By assumptions we in this technique understand designated unit (single literal) clauses passed along with the rest of the input to the solver. In case the input is unsatisfiable, the solver is able to report which of these units were actually used in the proof.

Solving under assumptions provides us with essentially free mechanism for computing small reasons. We simply designate the literals of $Lits(s)$ to be treated as unit assumptions in the query (2) above. In the unsatisfiable case, we obtain a reason $r \subseteq Lits(s)$ as required.

3.1.2 Explicit Minimization

The subset r of the assumption literals $Lits(s)$ returned by the solver can typically be further reduced. We can explicitly minimize it by trying to remove literals one by one. If the respective query remains unsatisfiable we leave the literal out. Otherwise we put it back. In a number of steps proportional to $|r|$ we obtain a final reason set $r^* \subseteq r$ minimal with respect to subset relation such that the query

$$SAT?[\, r^* \wedge T \wedge (L)' \,],$$

is unsatisfiable. Note that the order in which the literals are tried out influences the final result and may be subject to heuristical tuning. Although reason minimization is an expensive operation (we need one extra SAT-solver call per literal), experiments show that it is an important ingredient for solving hard problems.

3.1.3 Keeping the Reason Disjoint from G

As will become clear later, to ensure correctness of PDR we require that the computed reason r is not consistent with the goal formula G. In other words, the formula $r \wedge G$ must be unsatisfiable. This can be always achieved, because the algorithm never attempts to extend a goal state and thus we only start minimizing with unsatisfiable $\mathit{Lits}(s) \wedge G$.

A particularly simple strategy, which works whenever the goal formula G is in the form of a set of unit clauses (as is the case in planning), minimizes the reason as much as possible, but afterwards puts a single literal back to r provided the required unsatisfiability condition would be otherwise compromised. We look for a literal l to be added to r (unless found to be already present) such that $\{\neg l\} \in G$. This condition is, under our assumptions, both sufficient and necessary to ensure that $r \wedge G$ is unsatisfiable and can be established without additional calls to a SAT-solver.

Another option is to make sure beforehand that the set of goal states is included in its own preimage with respect to the transition relation T. For instance, making the transition relation reflexive by adding self-loops to every state achieves this without actually affecting the existence or the length of the shortest witnessing path. In planning, we can simply include the noop action into the action set.

3.2 Data Structures and Main Invariants

We continue our exposition of PDR by describing its main data structures. The clause sets representing the reachability information are organized in a sequence[3] $L_0, L_1, \ldots$ and we refer to them as *layers*. At any moment during the run of the algorithm the layers satisfy the following three invariants:

1) L_0 is equivalent to G,

2) $L_{j+1} \subseteq L_j$ and thus $L_j \Rightarrow L_{j+1}$ for any $j \geq 0$,

3) $(L_j)' \wedge T \Rightarrow L_{j+1}$ for any $j \geq 0$, i.e., L_{j+1} over-approximates the preimage of L_j.

When the algorithm starts, the layer L_0 is initialized to be equal to the set G and all the remaining layers are empty. Thus, initially, the invariants 1), 2), and 3) are trivially satisfied. We will return to the invariants at an appropriate place below to argue that they are indeed maintained by the algorithm.

The constructed paths – there can actually be more than one – are represented via so called *obligations*.[4] Formally, an obligation is a pair (s, i) consisting of a state s and

3. At any point in time only finitely many layers are non-empty and need to be represented in memory.
4. The full term is *proof obligation*. It comes from the verification perspective, where an obligation must be proven unreachable, otherwise the property of the system does not hold and a counter-example is found.

an index i. The index is a natural number denoting the position of s with respect to the layers. It may be seen to stand for a lower bound estimate on the distance of s towards the goal. In a practical implementation an obligation also stores a link to its parent, i.e. to the obligation from which it was derived, and we use such links to recover the full witnessing path once the goal has been reached.

3.3 The Algorithm

We are now ready to have a look at the overall structure of PDR (see Pseudocode 1). After initializing the layers (line 1), the algorithm proceeds in *iterations*, counted by a variable k. The main part of each iteration is a *path construction* phase during which the algorithm attempts to build a path step by step and terminates if a full witnessing path is actually discovered. If iteration k finishes without completing the path, PDR has established that there is no witnessing path for the transition system of length k or less.

As we describe in detail below, path construction can be enhanced by the *obligation rescheduling* technique, which allows the algorithm during iteration k to consider paths potentially longer than k. Path construction is in each iteration complemented by a *clause propagation* phase, which attempts to "push clauses" from low index layers to high index ones and checks for a global convergence within the layers, the occurrence of which implies that no witnessing path (of any length) is possible. Neither obligation rescheduling nor clause pushing are needed for ensuring correctness of the algorithm, but typically greatly improve its performance.

3.3.1 Path Construction

The path construction phase of iteration k starts by using a SAT-solver to pick an initial state s satisfying L_k (lines 4 and 5). It manipulates a set Q, working as a priority queue, for storing obligations. The set Q is initialized (line 6) with the obligation (s, k). The inner loop (starting at line 7) processes individual obligations, selecting first those that are estimated to be closer to the goal (line 8). Let (s, i) be a selected obligation. When $i = 0$, this means that a full witnessing path has been constructed and the algorithm terminates (line 10). The path extension query described previously is executed next (line 11). We look for a successor of s which would satisfy the clauses of L_{i-1}. Since s was originally obtained satisfying L_i, this represents an attempt to extend the current path one step closer towards the goal. If the extension is successful both the new obligation $(t, i-1)$ to be worked on next and the current (s, i) are stored in Q (lines 12, 13). In the opposite case, a new clause is derived from the reason of the failure and used to strengthen the layers $L_0, \ldots, L_i$ (lines 15, 16). Notice that after the strengthening the state s no longer satisfies L_i. This means the approximation has become strictly more precise, which ensures progress.

3.3.2 Obligation Rescheduling

The unsatisfiable branch of the extension attempt continues with two more lines (19, 20), which are not necessary for correctness of PDR. Without this *obligation rescheduling* technique, the algorithm would now forget the obligation (s, i) and would return to work on its parent. This would correspond to a strict backtracking behavior, with the set Q functioning as a stack. Instead, we try to reuse the obligation and reschedule it one step further from

Pseudocode 1 Algorithm $\texttt{PDR}(\Sigma, I, G, T)$:

Input:

A symbolic transition system $\mathcal{S} = (\Sigma, I, G, T)$

Output:

A witnessing path for $\mathcal{S}$ or a guarantee that no path exists

```
L_0 ← G; foreach j > 0 : L_j ← ∅ /* Initialize the layers */
for k = 0, 1, ... do
  /* Path construction: */
  while SAT?[ I ∧ L_k ] do
    extract state s from the model
    Q ← {(s, k)}
    while Q not empty do
      pop some (s, i) from Q with minimal i
      if i = 0 then
        return WITNESSING PATH FOUND
      if SAT?[ Lits(s) ∧ T ∧ (L_{i−1})' ] then
        extract a successor state t from the model
        Q ← Q ∪ {(s, i), (t, i − 1)}
      else
        compute a reason r ⊆ Lits(s) such that r ∧ G is unsatisfiable
        foreach 0 ≤ j ≤ i : L_j ← L_j ∪ {¬r}

        /* Obligation rescheduling: */
        if i < k then
          Q ← Q ∪ {(s, i + 1)}

  /* Clause propagation: */
  for i = 1, ..., k + 1 do
    foreach c ∈ L_{i−1} \ L_i do
      /* Clause push check */
      if not SAT?[ ¬c ∧ T ∧ (L_{i−1})' ] then
        L_i ← L_i ∪ {c}
    /* Convergence check */
    if L_{i−1} = L_i then
      return NO PATH POSSIBLE
```

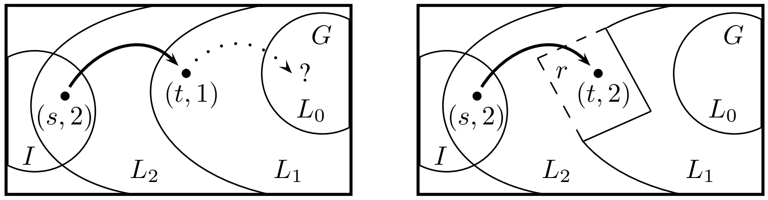

Figure 2: Layers, obligations and rescheduling.

the goal. This typically boosts the performance as it allows PDR to discover paths longer than the current iteration number k, before the existence of paths of length k is ruled out. Without obligation rescheduling, PDR only looks for optimal length paths.

The obligation rescheduling technique as well as the general interaction between layers, obligations, and reasons are illustrated in Figure 2. There, PDR is in the middle of the path construction phase of iteration 2. The algorithm is attempting to extend the obligation $(t, 1)$ and to reach a goal state in one step (left). When the attempt fails (right), PDR generalizes from t, obtains a reason r, and learns a new clause $c = \neg r$ to strengthen the layers L_1 and L_0. Notice how the obligation is rescheduled to $(t, 2)$ and PDR can now attempt to extend it and satisfy the new L_1 in one step. Without rescheduling, PDR would forget t and would go back to extending $(s, 2)$ instead.

3.3.3 Clause Propagation

Let us proceed to the clause propagation phase. It checks for every clause c lying in layer L_{i-1} but not in L_i (line 24) whether it could be pushed forward and added to strengthen the layer L_i. This is done when the query on line 26 returns UNSAT, which means that

$$(L_{i-1})' \wedge T \Rightarrow c,$$

and adding the clause to L_i will preserve invariant 3). The motivation for clause propagation is that stronger layers will provide for a better guidance in the following path construction phase. Indeed, notice that some clauses may even enter the till now empty layer L_{k+1}. More importantly, however, clause propagation is an opportunity to check for equality between two neighboring layers (line 29).[5] As we explain later, the equality implies that there is no witnessing path for the transition system of any length and so PDR terminates (line 30).

3.3.4 Example Execution

Let us recall the STS $\mathcal{S} = (\Sigma, I, G, T)$ from Example 1, defined over a two variable signature $\Sigma = \{p, q\}$ with the initial, goal, and transition formulas $I = \{\neg p\}$, $G = \{p, q\}$, and

$$T = \{p \vee \neg p' \vee \neg q',\ p \vee q \vee \neg p',\ p \vee q \vee q',\ p \vee \neg q \vee p',\ \neg q \vee \neg p' \vee \neg q'\},$$

5. It is not necessary to perform the relatively expensive clause pushing (which requires one SAT-solver call per clause) before checking for layer equivalence. Experience from model-checking suggests, however, that pushing substantially helps to speed up the convergence and clause propagation is one of the key ingredients for efficiently detecting unsatisfiable problems.

step	event / query	effect / explanation
1	Initialization	$L_0 = \{p, q\}$; $L_i = \emptyset$ for $i > 0$
2	Path construction (iteration 0)	$k = 0$
3	$SAT?[\, I \wedge L_0 \,]$ false	$\neg p \wedge p$ unsat.
4	Clause propagation	
5	$SAT?[\neg p \wedge T \wedge (L_0)' \,]$ false	$\neg p \wedge (p \vee \neg p' \vee \neg q') \wedge (p \wedge q)'$ unsat.
6	push p and add it to L_1	$L_1 = \{p\}$
7	$SAT?[\neg q \wedge T \wedge (L_0)' \,]$ true	$\neg q \wedge T \wedge (p \wedge q)'$ sat.
8	Path construction (iteration 1)	$k = 1$
9	$SAT?[\, I \wedge L_1 \,]$ false	$\neg p \wedge p$ unsat.
10	Clause propagation	
11	$SAT?[\neg q \wedge T \wedge (L_0)' \,]$ true	$\neg q \wedge T \wedge (p \wedge q)'$ sat.
12	$SAT?[\neg p \wedge T \wedge (L_1)' \,]$ true	$\neg p \wedge T \wedge p'$ sat.
13	Path construction (iteration 2)	$k = 2$
14	$SAT?[\, I \wedge L_2 \,]$ true	$\neg p$ sat.
15	extract a state and initialize Q	$s = \{p \mapsto \mathbf{0}, q \mapsto \mathbf{0}\}$; $Q = \{(s, 2)\}$
16	$SAT?[\, Lits(s) \wedge T \wedge (L_1)' \,]$ false	$\neg p \wedge \neg q \wedge (p \vee q \vee \neg p') \wedge (p)'$ unsat.
17	compute a reason and learn a clause	$r = \neg p \wedge \neg q$; $L_2 = \{p \vee q\}$, $L_1 = \dots$
18	$SAT?[\, I \wedge L_2 \,]$ true	$\neg p \wedge (p \vee q)$ sat.
19	extract a state and initialize Q	$t = \{p \mapsto \mathbf{0}, q \mapsto \mathbf{1}\}$; $Q = \{(t, 2)\}$
20	$SAT?[\, Lits(t) \wedge T \wedge (L_1)' \,]$ true	$\neg p \wedge q \wedge T \wedge (p)'$ sat.
21	extract a state and store it in Q	$u = \{p \mapsto \mathbf{1}, q \mapsto \mathbf{0}\}$; $Q = \{(t, 2), (u, 1)\}$
22	$SAT?[\, Lits(u) \wedge T \wedge (L_0)' \,]$ true	$p \wedge \neg q \wedge T \wedge (p \wedge q)'$ sat.
23	extract a state ...	$v = \{p \mapsto \mathbf{1}, q \mapsto \mathbf{1}\}$;
24	... and store it in Q	$Q = \{(t, 2), (u, 1), (v, 0)\}$
25	witnessing path found	return t, u, v

Table 1: Execution trace of PDR on the STS from Example 1.

respectively. Table 1 showcases an execution trace of PDR on this STS. It lists all the SAT-solver queries in order of execution, provides explanation for the results in the form of unsatisfiable cores, and tracks changes to the global variables. Notice that in step 6 the clause p is successfully pushed from layer L_0 to L_1. In step 17 the new clause $p \vee q$ is formally added to L_2, L_1, and L_0. However, it only properly strengthens the layer L_2.

3.4 Correctness

To show correctness of PDR we first review invariants 1)–3) and demonstrate that they are indeed preserved during the run of the algorithm. We than add a fourth observation, invariant 4), important for showing correctness in the unsatisfiable case. The invariants are then used to prove an independent lemma and, finally, also the main correctness theorem.

3.4.1 Four Invariants

Invariant 1) states that the layer L_0 is equivalent to G. This could be only violated when a new clause c is added to L_0 (line 16). But since $c = \neg r$ for some reason r and we assume that $r \wedge G$ is unsatisfiable (recall Section 3.1.3), we have $G \Rightarrow c$ and invariant 1) is preserved. Invariant 2) asserts that $L_{j+1} \subseteq L_j$ for any index $j \geq 0$. This is trivially maintained both when a new clause is added to the layers (line 16) and during clause pushing (line 27).

Invariant 3) is the statement that $(L_j)' \wedge T \Rightarrow L_{j+1}$ for any $j \geq 0$. When a new clause c is added to the layers $L_0, \ldots, L_i$ (line 16) there has been an unsuccessful extension of obligation (s, i), which means that $c = \neg r$ for some reason $r \subseteq \mathit{Lits}(s)$. By the definition of a reason we have that the formula $r \wedge T \wedge (L_{i-1})'$ is unsatisfiable and, therefore, $(L_{i-1})' \wedge T \Rightarrow c$. This guarantees that invariant 3) will hold for $j = i - 1$ after clause c is added to L_i. However, because the layers of index $j < i - 1$ are even stronger than L_{i-1} by invariant 2), invariant 3) will also hold for $j < i - 1$. Finally, the case of $j > i - 1$ is trivial. As already discussed (recall Section 3.3.3) invariant 3) is also preserved during clause propagation.

There is one more observation that we will need in order to show correctness of PDR. Let us call it invariant 4). Invariant 4) states that when the path construction of iteration k finishes there is no initial state satisfying L_k. This follows from the fact that the query on line 4 must be unsatisfiable for the path construction to finish.

3.4.2 Correctness and Termination

Lemma 1. *When PDR creates (either on line 6 or on line 13) a new obligation (s, i) then $s \models L_i$. Moreover, $s \not\models L_j$ for any $j < i$. This latter property is maintained throughout the run of the algorithm and, in particular, holds also after rescheduling (line 20).*

Proof. First note that it is sufficient to show the second part only for $j = i - 1$ and then use invariant 2). Also note that during the run of PDR clauses are only *added to* and never *removed from* the layers. This means it is sufficient to focus on the moments when a new obligation is created: if $s \not\models L_j$ when the obligation (s, i) is created, this must also hold later, after the layer L_j has been strengthened by addition of new clauses.

Let us now consider iteration k. When creating a new obligation (s, k) on line 6, we have $s \models L_k$ by construction and $s \not\models L_{k-1}$ by invariant 4). When creating a new obligation $(t, i - 1)$ on line 13, we assume that its parent (s, i) already satisfies our lemma and, in particular, that $s \not\models L_{i-1}$. We now have $t \models L_{i-1}$, again by construction, and if $i > 1$ we infer $t \not\models L_{i-2}$ from our assumption about s and from invariant 3). Finally, an obligation (s, i) is only rescheduled to $(s, i + 1)$ after the addition of a clause $c = \neg r$ into L_i for some $r \subseteq \mathit{Lits}(s)$. This means that $s \not\models L_i$ at the time of the rescheduling. □

Lemma 1 captures the intuition that a state s of an obligation (s, i) is always at least i steps from reaching the goal. It follows from this lemma that PDR never attempts to extend a goal state, which is the assumption we relied on in Section 3.1.3 to show that we can always keep reasons disjoint from the goal formula G.

Theorem 1 (Bradley, 2011)**.** *Given an STS $\mathcal{S} = (\Sigma, I, G, T)$ the algorithm terminates and returns a witnessing path for $\mathcal{S}$ if and only if $\mathcal{S}$ is satisfiable.*

Proof. It is easy to see that if PDR returns a path (line 10)[6] it is a witnessing path for $\mathcal{S}$. Indeed, for every considered obligation (s, i) the state s is reachable from an initial state and when $i = 0$ the state s satisfies L_0, which is equivalent to G by invariant 1).

If PDR terminates claiming that no witnessing path exists (line 30) the path construction phase of iteration k has finished and there is an index $0 \leq j \leq k$ such that $L_j = L_{j+1}$. By combining invariants 1)–3) and the detected equality we obtain $G \Rightarrow L_j$ and $(L_j)' \wedge T \Rightarrow L_j$. This together with invariant 4) rules out the existence of a witnessing path of any length.[7]

To address termination we first show that the path construction phase of iteration k cannot run indefinitely. Recall that PDR always selects for extension an obligation with minimal index i (line 8). Thus it follows from Lemma 1 that after a successful extension of obligation (s, i) the new extracted state t is not equal to any other state previously considered during iteration k (t is currently the only state that satisfies L_{i-1}). On the other hand, after an unsuccessful extension of obligation (s, i) the addition of the new clause c to the layer L_i ensures that $s \not\models L_i$ anymore (recall that $c = \neg r$ for some $r \subseteq \mathit{Lits}(s)$). This means there cannot be more than $k \cdot 2^{|\Sigma|}$ repetitions of the "Q-processing" while-loop (line 7) and more than $2^{|\Sigma|}$ repetitions of the outer while-loop of path construction (line 4).

We are left to bound the maximal number of iterations of PDR. By invariant 2) the sets of states represented by the individual layers are ordered by inclusion and after the clause propagation phase of iteration k finishes the first $k+1$ of these sets are necessarily distinct. Thus there cannot be more than $2^{|\Sigma|}$ iterations before PDR terminates. □

3.5 Inductive Reason Minimization

Inductive reason minimization is a technique for obtaining small reasons from unsuccessful extensions. We postponed discussing the technique after the presentation of PDR, because it relies in an non-obvious way on the overall architecture of the algorithm. Moreover, although "inductiveness" was one of the main initial ideas behind PDR (Bradley, 2011), our experiments suggest that the practical value of this relatively advanced technique for automated planning may be limited.

To demonstrate inductive minimization let us once more recall the situation of an unsuccessful extension of obligation (s, i). We want to compute a reason r, which is a subset of $\mathit{Lits}(s)$, ideally as small as possible, such that the formula

$$r \wedge T \wedge (L_{i-1})' \tag{3}$$

is unsatisfiable. Now, the crucial observation is that since we are in the next step going to strengthen the layers $L_0, \ldots, L_i$ (and, in particular, the layer L_{i-1}) with the clause $c = \neg r$, we may already assume c on the "primed" side of (3) when minimizing r. This means, we can use the stronger query

$$r \wedge T \wedge (L_{i-1} \wedge \neg r)'.$$

Having r on both sides of the transition breaks monotonicity: as r gets weaker, $\neg r$ gets stronger. Satisfiable query may become unsatisfiable again when more literals are removed

6. Line 10 of Pseudocode 1 only reports the existence of a path. A true witnessing path can, however, be easily recovered by following the parent pointers (see Section 3.2) from the last obligation $(s, 0)$.

7. The layer L_j can be understood as the promised "certificate of non-reachability". It is a property of the goal states incompatible with the initial formula that is preserved by traversing the transitions backwards.

from r. This makes the task of finding subset-minimal "inductive reason" computationally difficult (Bradley & Manna, 2007).

Pseudocode 2 Inductive Reason Minimization:

Input:

A set of clauses L and a cube r such that
the formulas $r \wedge T \wedge (L)'$ and $r \wedge G$ are unsatisfiable

Output:

Minimized inductive reason $r^* \subseteq r$, i.e.,
the formulas $r^* \wedge T \wedge (L \wedge \neg r^*)'$ and $r^* \wedge G$ are unsatisfiable

repeat
  $r_0 \leftarrow r$
  foreach $l \in r$ **do** /* Check each literal of r_0 once */
    if there is $l_0 \in (r_0 \setminus \{l\})$ such that $\{\neg l_0\} \in G$ **then** /* Can try removing l */
      $r_0 \leftarrow (r_0 \setminus \{l\})$
      if $SAT?[\, r_0 \wedge T \wedge (L_{i-1} \wedge \neg r_0)' \,]$ **then**
        $r_0 \leftarrow (r_0 \cup \{l\})$ /* Put the literal back */
until $r = r_0$ /* No removal in the last iteration */

return r

In Pseudocode 2 we present a simple version of inductive reason minimization with no minimality guarantee, which was, however, successfully applied in hardware model checking (Eén et al., 2011). The procedure is meant to improve on and replace the explicit reason minimization described in Section 3.1.2. It assumes that the goal formula G is in the form of a set of unit clauses to keep the reason disjoint from G (see Section 3.1.3). Notice that in the non-monotone setting of inductive minimization it makes sense to retry all the literals once a single literal has been successfully removed. That is why the procedure employes the outer loop to continue minimizing till a true "fixed point" is reached.

3.6 Notes on Implementation

There are several important points relevant for a practical implementation of PDR which did not fit the level of detail of the presented pseudocode. We moved them to this section.

3.6.1 Representation of the Layers

Because the individual clause sets L_i are ordered by inclusion it is advantageous to store each clause only once, namely at the position where it appears last. This convention has been named *delta encoding* by Eén et al. (2011). It is defined by setting

$$\Delta_i = L_i \setminus L_{i+1} \qquad \text{and} \qquad L_i = \textstyle\bigcup_{j \geq i} \Delta_j .$$

With delta encoded layers clause propagation moves clauses around instead of copying them and the equality check between neighboring layers becomes an emptiness check of the respective delta.

3.6.2 Clause Subsumption

It has been observed that PDR often derives a clause c into a layer L_i while a weaker clause is already present. It pays off to remove from Δ_i (so effectively from $L_0, \dots, L_i$) all those clauses which are *subsumed* by this new clause c. Keeping the layers small this way (while preserving their semantic strength) helps to speed up the algorithm in several places. It reduces the load on the SAT-solver (provided we can retract the subsumed clause from it) and it also means fewer clauses need to be checked for pushing.

3.6.3 Obligation Subsumption

Another place where subsumption can (and should) be employed is when dealing with obligations. It may happen that before an obligation (s, i) is handled, the layer L_i, where the obligation "belongs", gets in the meantime strengthened in such a way that s no longer satisfies L_i. At this point we already know that the obligation cannot be extended, so we can save one SAT-solver call and directly reschedule the obligation. Such a situation can be detected by subsumption: a state s does not satisfy a clause set L if and only if there is a clause $c \in L$ such that $c \subseteq \neg Lits(s)$. We insert the test at the point in the algorithm where a new clause c is derived into L_i. We then check for subsumption all the obligations of the form (s, i) that are currently in the set Q.

3.6.4 Breaking Ties when Popping from Q

When popping obligations from the set Q (line 8) we make sure we select among those estimated closest to the goal. This is necessary for ensuring termination of the algorithm. Otherwise, however, we are free to choose any obligation with the minimal index i. Two prominent strategies for resolving this "don't-care" non-determinism are

- to select the most recently added obligation first, which we call the *stack* strategy,
- to select the least recently added obligation first, the *queue* strategy.

The stack strategy prefers exploring longer paths before short ones, while the queue strategy does the opposite. Eén et al. (2011) report a small performance gain with the stack strategy on hardware model checking benchmarks. We used the stack strategy as the default in our experiments and observed its superiority over the queue strategy in satisficing planning, but also its slightly unfavorable effect on plan quality (see Section 5.3.3).

4. PDR without a SAT-solver

Although it is possible to encode a STRIPS planning problem into an STS and use a general implementation of PDR to solve it, a more efficient approach can be adopted. The approach relies on an observation that the work normally delegated in PDR to the SAT-solver can in the case of planning with the sequential plan semantics be instead implemented directly by a planning-specific procedure. Not only do we gain with this procedure a polynomial time guarantee for the response of each extension query, but the ensuing perspective also enables us to devise new improvements of the overall algorithm.

The SAT-solver is employed in several places within PDR. We will start by focusing on its primary role which lies in extending the current path by one step. In Section 4.1 we develop procedure `extend` to replace the SAT-solver in path extension queries. A separate section is then devoted to discussing inductive reason minimization in the planning context. In Section 4.3 we deal with replacing the remaining SAT-solver calls. We show how to efficiently implement clause pushing for positive STRIPS planning problems, a subclass of STRIPS problems that is typically used in practice. We then discuss the possibility of reversing the default search direction of PDR in Section 4.4 and, finally, we propose several improvements of the algorithm in Section 4.5.

4.1 Planning-Specific Path Extensions

Let us recall the interface for path extensions, which is normally implemented in PDR by a call to a SAT-solver (Section 3.1). Given a state s and a set of clauses L decide whether there exists a state t, a successor of s with respect to the transition relation T, such that t satisfies L. In the positive case, which we refer to as a *successful* extension, return such a t. In the negative case, when no such a successor exists, compute a reason r for the failure in the form of a preferably small subset of the literals defining s, such that no state satisfying r has a successor that would satisfy L.

Let us assume a STRIPS planning problem $\mathcal{P} = (X, s_I, g, \mathcal{A})$ is given. We now gradually work towards a planning-specific implementation of the above interface within the procedure $\texttt{extend}(s, L)$. Our central idea is to emulate the mechanics of the sequential encoding $\mathcal{S}_{\mathcal{P}}^{seq}$ (see Section 2.6). This makes the implementation particularly straightforward from the perspective of the positive part of the interface. Given a state s, we can simply iterate over all the actions $a \in \mathcal{A}$, generate a successor $t_a = \mathit{apply}(s, a)$ whenever a is applicable in s, and check for each t_a whether it satisfies the clauses of L. If such a successor is found, it is returned and the procedure terminates. Such an iteration is clearly affordable from the complexity point of view. In fact, it is very similar in spirit to what all explicit state planners need to do: they enumerate successor states and evaluate their heuristic value.

The non-trivial part of the `extend` procedure deals with computing a small reason in the case of an unsuccessful extension. We conceptually simplify the problem by first separately collecting a set of reasons R_a for every action $a \in \mathcal{A}$ and then computing the overall reason r as a union

$$r = \bigcup_{a \in \mathcal{A}} r_a \tag{4}$$

of reason contributions $r_a \in R_a$ selected in a way that minimizes the size of the union. The idea is that each $r_a \in R_a$ is a distinct reason for why the action a cannot be applied in s to produce a successor state t that would satisfy all the clauses from L. The union (4) then justifies why there is no such a successor state via any action $a \in \mathcal{A}$ whatsoever.

In the rest of this section, we first explain how the individual reasons $r_a \in R_a$ for an action $a \in \mathcal{A}$ are derived from the action's failed preconditions and from those clauses of L which the respective successor state fails to satisfy. We then show how this reason collecting process can be in practice sped up by employing certain subsumption concepts. Finally, we present our approach to obtaining a small overall reason r, along with a detailed pseudocode of the `extend` procedure and a proof of its correctness. To satisfy the requirement of PDR that the final reason be disjoint from the set of goal states, we here adopt the solution of formally adding the noop action to the action set (see Section 3.1.3).

4.1.1 Reasons for Individual Actions

We construct the set of reasons R_a for a particular action a as follows. First we check whether the action a is applicable in the given state s. If not then there is a precondition literal $l \in \mathit{pre}_a$ false in s. The negation of each such literal represents a singleton reason $\{\neg l\} \subseteq \mathit{Lits}(s)$ which we add to R_a. Clearly, as long as a state satisfies $\neg l$ there is no way a can be used to produce a successor state, let alone one that would satisfy L.

Next, we compute the successor state $t_a = \mathit{apply}(s, a)$. Strictly speaking, t_a cannot be regarded as a true successor if a is not applicable in s. Nevertheless, t_a is useful even then for computing further reasons, namely reasons corresponding to clauses of L that are false in t_a. These are either clauses that were already false in s and a failed to make them true or clauses that became false due to an effect of a. For each such clause c we add to R_a a reason r_c consisting of negations of literals $l \in c$. As an optimization, we only include those negated literals which were not made false by an effect of a. Since the other literals will always be false after a is applied due to its effects, as long as s satisfies r_c, the successor t_a cannot satisfy c. Summarizing formally, this is the final set of reasons we obtain:

$$R_a = \{\{\neg l\} \mid l \in \mathit{pre}_a \text{ and } s \not\models l\} \cup \{r_c \mid c \in L \text{ and } t_a \not\models c\},$$

where $r_c = \{\neg l \mid l \in c \text{ and } \neg l \notin \mathit{eff}_a\}$. It is easy to check that $r_c \subseteq \mathit{Lits}(s)$ as required. Notice that the set R_a is empty if an only if the action a is applicable in s and the successor t_a satisfies all the clauses from L.

Example 2. Starting from a state $s = \{o \mapsto \mathbf{0}, p \mapsto \mathbf{0}, q \mapsto \mathbf{0}, r \mapsto \mathbf{0}\}$, let us compute the reasons for an action $a = (\mathit{pre}_a, \mathit{eff}_a)$ with $\mathit{pre}_a = \{\neg p, q\}$ and $\mathit{eff}_a = \{o, \neg r\}$ with respect to the clause set $L = \{o \vee q, p \vee r\}$. Because the precondition q is not satisfied in s, one reason is $\{\neg q\}$. Next we compute $t_a = \mathit{apply}(s, a) = \{o \mapsto \mathbf{1}, p \mapsto \mathbf{0}, q \mapsto \mathbf{0}, r \mapsto \mathbf{0}\}$. The first clause, $o \vee q$ is satisfied in t_a and so does not give rise to a reason. The second clause, $p \vee r$, however, is false in t_a. The reason corresponding to the second clause is $\{\neg p\}$. The other negated literal, $\neg r$, is not part of the reason, because it was explicitly set to false by an effect of a. The final reason set R_a we obtain is thus $\{\{\neg p\}, \{\neg q\}\}$. Notice that both the computed reasons are subsets of $\mathit{Lits}(s) = \{\neg o, \neg p, \neg q, \neg r\}$.

Correctness of the reason set construction is captured by the following lemma.

Lemma 2. *Let $r_a \in R_a$ be any reason for an action $a \in \mathcal{A} \cup \{\mathit{noop}\}$ as defined above. Then*

$$r_a \wedge AP \wedge AE \wedge CF \wedge (L)' \models \neg p_a,$$

where AP, AE and CF are, respectively, the action precondition, the action effect and the classical frame axioms used in the transition formula T^{seq} of the sequential encoding $\mathcal{S}^{seq}_{\mathcal{P}}$.

Proof. Let us first assume that $r_a = \{\neg l\}$ is a reason derived from a failed precondition literal $l \in \mathit{pre}_a$. There is an action precondition axiom $\neg p_a \vee l \in AP$ from which the conclusion $\neg p_a$ follows by a single resolution inference with the unit assumption $\neg l$.

The other possibility is that $r_a = \{\neg l \mid l \in c \text{ and } \neg l \notin \mathit{eff}_a\}$ for some clause $c \in L$ false in the successor state t_a. There must be an action effect axiom $\neg p_a \vee \neg l' \in AE$ for every literal $l \in c$ such that $\neg l \in \mathit{eff}_a$ and also a classical frame axiom $\neg p_a \vee l \vee \neg l' \in CF$ for every literal $l \in c$ such that $\neg l \notin \mathit{eff}_a$ (if the literal l was in eff_a the clause c would be satisfied in

t_a). By resolving these axioms on the respective primed literals $\neg l'$ with the primed version $(c)' \in (L)'$ of the clause c we obtain a clause $\neg p_a \vee \bigvee_{l \in r_a} \neg l$ from which the final unit clause $\neg p_a$ can be derived by resolution with the available assumptions from r_a. □

4.1.2 Reason Subsumption

Before we describe how to compute the overall reason r from the actions' contributions R_a, let us note that there are two useful notions of subsumption both between individual reasons and between reason sets, which can be used to simplify the reason sets before the computation is started. The subsumption between individual reasons inside one particular R_a is simply the subset relation. It does not make sense to keep both r_1 and r_2 inside R_a if $r_1 \subseteq r_2$. Keeping the smaller r_1 is sufficient, because whenever we would decide to pick r_2 as the reason for a inside the union (4), switching to r_1 instead could only make the result smaller. In practice, we only check for this kind of subsumption between the unit reasons of failed preconditions and the reasons from the false clauses.[8] This can be implemented by simply ignoring those false clauses that would have been true if the action was applicable.

Dually to the above, we can discard the whole reason set R_a of an action a if there is another action b with reason set R_b such that

$$\forall r_b \in R_b \; \exists r_a \in R_a \; r_a \subseteq r_b.$$

Here we remove the reason set R_a, which is in some sense more lean, because for any contribution $r_b \in R_b$ there is a choice for $r_a \in R_a$ which would be dominated by r_b in the final union r. For efficiency reasons, we only exploit this trick in our implementation with respect to one particular action in the role of the "subsuming" action b, namely the noop action. As mentioned before, we include the noop action to the action set to ensure PDR's correctness. Its reason set R_{noop} consists of reasons corresponding to those clauses from L which are false in s.[9] If an action a does not make any of these clauses true in its corresponding successor state, its reason set R_a will be subsumed in the described sense by R_{noop} and can be skipped.

4.1.3 Computing the Overall Reason

Computing the overall reason r amounts to selecting for every a a particular reason $r_a \in R_a$ such that union (4) is as small as possible. Stated in this general form we are facing an optimization version of an NP-complete problem. In fact, it is easily seen to be a dual of the Maximum Subset Intersection problem shown NP-complete by Clifford and Popa (2011). We therefore do not attempt to find an optimal solution for it and contend ourselves with a reasonable approximation instead.

We sort the reason sets R_a according to their size $|R_a|$ and traverse them from smaller to larger ones. The idea is to deal with the more constrained cases first before moving on to those where we have more freedom. During the traversal, we maintain an unfinished union r_0 which is initialized as the empty set $\emptyset$. Then each reason set R_a is considered in turn and we pick from each a reason $r_a \in R_a$ that minimizes the size of $r_0 \cup r_a$ and update

8. This is sufficient, because PDR keeps layers L subsumption reduced and so the reasons for false clauses are subsumption reduced for free.
9. PDR only calls `extend`(s, L) when $s \not\models L$, so there is always at least one such clause.

the set r_0 accordingly to describe the union of those reasons selected so far. Although this greedy pass through the action sets does not guarantee that the final value of r_0 is minimal, it already gives satisfactory results.

To improve the quality of the reason set even further, we then minimize r_0 with respect to the subset relation by explicitly trying to remove individual literals and checking whether the result is still a valid overall reason. This is a direct adaptation of the explicit reason minimization procedure employed in the original PDR (recall Section 3.1.2). In detail, we iteratively pick a literal $l \in r_0$ and check for every action a whether there is a reason $r_a \in R_a$ such that $r_a \subseteq (r_0 \setminus \{l\})$. If this is indeed the case, r_0 can be shrunk to $(r_0 \setminus \{l\})$, otherwise we continue with the old r_0 and try another literal instead. When all the literals have been tried out, we obtain the final result r.

4.1.4 Pseudocode and Correctness

The code of the procedure `extend`(s, L) is detailed in Pseudocode 3. The corresponding reason construction proceeds in three stages. In the first stage we collect reasons from the individual actions, constructing the sets R_a. This is performed during the same iteration through the action set which establishes whether a successor state t satisfying L exists. It either terminates by discovering such a t or computes a non-empty set R_a for every action a. The first stage also includes the subsumption-based filtering of reasons, both within a particular action's reason set and between the reason sets of the noop action and one other action. In the second stage, the above described simple greedy pass through the sets R_a computes an initial overall reason, which is then explicitly minimized in stage three.

Correctness of the `extend` procedure in the positive case as well as the fact that for any returned reason r we have $r \subseteq \mathit{Lits}(s)$ are easy to establish. The remaining argument is captured by the following lemma.

Lemma 3. *Let r be a cube returned by the procedure* `extend`(s, L)*. Then the formula*

$$r \wedge T^{seq} \wedge (L)' \tag{5}$$

is unsatisfiable, where T^{seq} is the transition formula of the sequential encoding $\mathcal{S}^{seq}_{\mathcal{P}}$.

Proof. We first observe that for every action $a \in \mathcal{A} \cup \{noop\}$ there is a reason r_a such that $r = \bigcup_{a \in \mathcal{A} \cup \{noop\}} r_a$. For those actions a for which $R_a \in \mathcal{R}$ this reason is initially picked during stage two (line 22) and possibly later changed to the reason r_a for which $r_a \subseteq (r \setminus \{l\})$ during stage three (line 26). For those actions whose reason set is subsumed by R_{noop} (line 14) we can formally pick the same reason as for noop.

Since $r_a \subseteq r$ for every action $a \in \mathcal{A} \cup \{noop\}$, we can use Lemma 2 to infer that formula (5) entails the unit clause $\neg p_a$ for every $a \in \mathcal{A} \cup \{noop\}$. But because formula (5) also trivially entails the at-least-one axiom $alo = \bigvee_{a \in \mathcal{A} \cup \{noop\}} p_a$, it must be unsatisfiable. □

It is easy to see that the procedure `extend`(s, L) runs in time polynomial in $|X|$, the number of state variables, $|\mathcal{A}|$, the number of actions of the planning problem, and $|L|$, the size of the given clause set. This is mainly enabled by the fact that `extend` emulates

Pseudocode 3 Procedure `extend`(s, L):

Input:

State s; a set of clauses L such that $s \not\models L$

Output:

Either state t, a successor of s such that $t \models L$
or a reason $r \subseteq \mathit{Lits}(s)$ such that no state satisfying r has a successor satisfying L

/* Stage one: look for the successor state and prepare the reason sets */
$L^s \leftarrow \{c \in L \mid s \not\models c\}$ /* Clauses false in s */
$R_{noop} \leftarrow \{\neg c \mid c \in L^s\}$ /* The reasons for the noop action */
assert $R_{noop} \neq \emptyset$ /* Follows from the contract with the caller */
$\mathcal{R} \leftarrow \{R_{noop}\}$ /* The set of reason sets collected so far */

foreach $a \in \mathcal{A}$ **do**
    $\mathit{pre}^s_a \leftarrow \{l \in \mathit{pre}_a \mid s \not\models l\}$ /* Preconditions false in s */
    $t \leftarrow \mathit{apply}(s, (\emptyset, \mathit{eff}_a))$ /* Ignore the preconditions and apply the effects of a */
    $L^t \leftarrow \{c \in L \mid t \not\models c\}$ /* Clauses false in t */
    if $\mathit{pre}^s_a = \emptyset$ **and** $L^t = \emptyset$ **then**
        return t /* The positive part: returning a successor */
    else if $L^s \subseteq L^t$ **then**
        pass /* Do nothing: the reason set would be subsumed by R_{noop} */
    else
        $L^t_0 \rightarrow \{c \in L^t \mid c \cap \mathit{pre}^s_a = \emptyset\}$ /* False clauses with a non-subsumed reason */
        $R_a \leftarrow \{\{\neg l\} \mid l \in \mathit{pre}^s_a\} \cup \{\{\neg l \mid l \in c \text{ and } \neg l \notin \mathit{eff}_a\} \mid c \in L^t_0\}$
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{R_a\}$ /* Record the reason set */

/* Stage two: compute an overall reason */
$r \leftarrow \emptyset$
foreach $R_a \in \mathcal{R}$ ordered by $|R_a|$ from small to large **do**
    pick $r_a \in R_a$ such that $|r \cup r_a|$ is minimal
    $r \leftarrow r \cup r_a$

/* Stage three (optional): minimize the reason */
foreach $l \in r$ **do**
    if for every $R_a \in \mathcal{R}$ there is $r_a \in R_a$ such that $r_a \subseteq (r \setminus \{l\})$ **then**
        $r \leftarrow (r \setminus \{l\})$

return r /* The negative part: returning a (subset minimal) reason cube */

the sequential encoding $\mathcal{S}_{\mathcal{P}}^{seq}$ and the individual actions can be in the first stage considered independently.[10]

A similar complexity guarantee seems to be achievable within a general-purpose SAT-solver when supplied with the same encoding and configured to prefer branching on the action variables A and setting them first to true. However, the inherent overhead connected with explicitly generating all the corresponding axioms and storing them in memory will be probably noticeable in practice. Moreover, the reason set subsumption optimization does not have a counterpart in a general-purpose solver.

4.2 Inductive Reason Minimization in Procedure `extend`

Inductive minimization is based on the idea that when checking whether a particular literal l can be removed from the final reason r we can assume that the clause $c = \neg r_0$ corresponding to the reduced reason $r_0 = (r \setminus \{l\})$ is already present in the set of clauses L (see Section 3.5). We can perform inductive minimization within the `extend` procedure by speculating for each action a whether we would be able to satisfy the additional clause c by applying a. Only if the answer is positive do we need to look for a "proper" reason $r_a \in R_a$.

Pseudocode 4 Stage three of `extend`(s, L); inductive version:

```
repeat
  r_0 ← r
  foreach l ∈ r do /* Check each literal of r_0 once */
    if there is l_0 ∈ (r_0 \ {l}) such that ¬l_0 ∈ g then /* Can attempt to remove l */
      r_0 ← (r_0 \ {l})
      foreach a ∈ 𝒜 do
        if for every l_0 ∈ eff_a : ¬l_0 ∉ r_0 then
          continue /* a passed by the inductive argument */
        if R_a ∈ ℛ and there is r_a ∈ R such that r_a ⊆ r_0 then
          continue /* a passed; it has its own small reason */
        if R_a ∉ ℛ and there is r_a ∈ R_noop such that r_a ⊆ r_0 then
          continue /* R_a was subsumed by R_noop which contains a small reason */

        /* Action a says: "Literal l cannot be removed" */
        r_0 ← (r_0 ∪ {l}) /* Put the literal back */
        break
until r = r_0 /* No removal in the last iteration */

return r
```

The idea is demonstrated in Pseudocode 4, which should be regarded as a replacement for stage three of the original `extend` procedure. Notice that we no longer consider the noop action to be part of the action set[11] and thus we need to explicitly check that there

10. When devising an analogous procedure for a parallel plan semantics, one would in general need to consider every subset of actions that can be applied together. This seems to make a polynomial time solution much more difficult, if not hopeless. However, see Section 6 for an interesting connection.

11. The noop action trivially passes the inductiveness check, because it can never make any clause true.

remains at least one literal incompatible with the goal condition g (line 4). There can still be actions, however, whose reason set has been subsumed by R_{noop} and for these we look for a reason in R_{noop} (line 11) whenever they fail to pass the inductiveness check (line 7). To avoid confusion we remark that the **continue** and **break** commands refer to the innermost cycle, which iterates over actions (line 6). Finally, we note that the presence of a small reason in R_{noop} depends only on the current value of r_0 and so the corresponding check could be precomputed outside the inner cycle.

Example 3. Recall Example 2, in which the action $a = (\{\neg p, q\}, \{o, \neg r\})$ failed in the state $s = \{o \mapsto \mathbf{0}, p \mapsto \mathbf{0}, q \mapsto \mathbf{0}, r \mapsto \mathbf{0}\}$ to provide a successor state that would satisfy the clauses from $L = \{o \vee q, p \vee r\}$ and so we computed a reason set $R_a = \{\{\neg p\}, \{\neg q\}\}$. Assume that apart from a the action set $\mathcal{A}$ contains just one other action, namely $b = (\{\neg r\}, \{p\})$, for which we obtain a reason set $R_b = \{\{\neg o, \neg q\}\}$. The overall reason after stage two is thus necessarily $r = \{\neg o, \neg q\}$. Assuming that the goal condition of the given problem is $g = \{o, p, q, r\}$, inductive minimization of the reason r could proceed as follows.

First we try the reason $r_0 = \{\neg o\}$. Since $o \in \mathit{eff}_a$ we cannot use the inductive argument for the action a and also no proper reason $r_a \in R_a$ has the property that $r_a \subseteq r_0$. Thus the literal $\neg q$ cannot be removed from r. Next we try the reason $r_0 = \{\neg q\}$. Since neither the action a nor b contain the literal q in their effect lists, the smaller reason is justified inductively for both actions and the overall reason r is reduced to $\{\neg q\}$. We cannot minimize r further, because there has to remain at least one literal $l_0 \in r$ such that $\neg l \in g$.

Looking from the perspective of the final learned clause $c = \neg r$ we observe that inductive minimization allows us (as in the example above) to remove from c every literal that cannot be made true by any action of the action set $\mathcal{A}$. Although this may seem like a powerful (global) criterion, it is effectively made redundant in practice by the so called *relaxed reachability analysis* (see Hoffmann & Nebel, 2001, Section 4.3), a standard preprocessing step which, before the actual search is started, removes from the problem all such unattainable variables as well as all actions that mention them in their precondition lists. Non-trivial invocations of inductive minimization were actually quite rare in our experiments.

4.3 Replacing the Remaining SAT-solver Calls

Beside the query for extending states, there are two other points in the formulation of PDR (recall Pseudocode 1 on page 275) where a SAT-solver call is employed. It is used to pick initial states at the beginning of the path construction phase (line 4) and is also central to verifying the condition for pushing clauses during the clause propagation phase (line 26). In planning, we can easily do without a SAT-solver in the first case, because there is only one initial state to be picked, namely the state s_I, and we just need to verify that s_I satisfies the clauses of L_k before the path construction phase of iteration k can be started.

We have basically two options how to deal with the second case. Since clause pushing is not need for ensuring correctness of PDR, we can simply leave the operation out. As we later show in our experiments, this does not significantly affect the performance on planning benchmarks, which are typically satisfiable. As a second option, we propose to restrict the planning formalism such that the query corresponding to a push check of a clause c, i.e.,

$$SAT?[\, \neg c \wedge T \wedge (L)' \,], \tag{6}$$

can be decided in polynomial time.[12] We say that a STRIPS planning problem is *positive* if the precondition list of every action and the goal condition of the problem consist of positive literals only.[13] It is easy to see that when running on a positive STRIPS problem, PDR only deals with positive clauses. The unit clauses of layer L_0, which describe the goal, are positive by assumption and all the learned clauses are transitively built only from the goal literals and from the action precondition literals. This observation allows us to reduce query (6) to the evaluation of "the positive part of the interface for path extensions".

Lemma 4. *Let $\mathcal{P} = (X, s_I, g, \mathcal{A})$ be a positive STRIPS planning problem and T^{seq} the transition formula of the sequential encoding $\mathcal{S}^{seq}_{\mathcal{P}}$. Further, let L be a set of positive clauses over X, c a positive clause over X, and $s_c : X \to \{\mathbf{0}, \mathbf{1}\}$ a state defined for every $p \in X$ by*

$$s_c(p) = \begin{cases} \mathbf{0} & \textit{if } p \in c, \\ \mathbf{1} & \textit{otherwise.} \end{cases}$$

Then the following formula

$$F_c = \neg c \wedge T^{seq} \wedge (L)'$$

is satisfiable if and only if there is an action $a \in \mathcal{A}$ such that $s_c \models pre_a$ and $apply(s_c, a) \models L$.

Proof. Let us first assume that there is an action $a \in \mathcal{A}$ applicable in s_c such that the successor state $t = apply(s_c, a)$ satisfies the clauses from L. Notice that $Vars(F_c) = X \cup A \cup X'$, where $A = \{p_a \mid a \in \mathcal{A}\}$ is the set of variables used for encoding applied actions. We define the following assignment $\alpha_a : A \to \{\mathbf{0}, \mathbf{1}\}$:

$$\alpha_a = \{p_a \mapsto \mathbf{1}\} \cup \{p_b \mapsto \mathbf{0} \mid b \in \mathcal{A}, b \neq a\}.$$

It is easy to verify that the joint assignment $(s_c \cup \alpha_a \cup t')$ satisfies F_c.

For the opposite direction, let us assume that an assignment $V : X \cup A \cup X' \to \{\mathbf{0}, \mathbf{1}\}$ satisfies the formula F_c. We fix an action $a \in \mathcal{A}$ such that $V(p_a) = \mathbf{1}$. Such an action must exist, because V satisfies the at-least-one axiom $alo = \bigvee_{a \in \mathcal{A}} p_a$, which is part of T^{seq}. By restricting V, first, to the state variables X, and, second, to the primed variables X', we extract, respectively, a state $s = V \restriction X$ and a state t such that $t' = V \restriction X'$. The axioms of T^{seq} ensure that the action a is applicable in s and that $t = apply(s, a)$.

We now notice that $s \models \neg c$, which means that $s(p) = \mathbf{0}$ for every $p \in c$. Thus if there is a difference between the states s and s_c it is only because of variables $p \notin c$ for which $s(p) = \mathbf{0}$ and $s_c(p) = \mathbf{1}$. But this means, for one thing, that since the action a is applicable in s, it must also be applicable in s_c (preconditions are positive) and, for the other, since $t \models L$, the successor state $t_c = apply(s_c, a)$ corresponding to s_c must also satisfy the clauses from L (the implication $\forall p \in X : s(p) = \mathbf{1} \Rightarrow s_c(p) = \mathbf{1}$ is preserved by the transition and becomes $\forall p \in X : t(p) = \mathbf{1} \Rightarrow t_c(p) = \mathbf{1}$ and the clauses from L are positive by assumption). □

12. In our current setting, there does not seem to be a general polynomial solution. In fact, even in the degenerate case of T encoding a transition by the single noop action and c being the empty clause, the query (6) boils down to satisfiability of L and its evaluation is thus an NP-complete problem.
13. Most of the standard planning benchmarks are positive STRIPS. Moreover, there is a well-known reduction (Gazen & Knoblock, 1997) that turns a general STRIPS problem into a positive one. The reduction introduces a new variable $\bar{p}$ for every variable p that occurs negatively in a precondition or in the goal and updates the actions to always force $\bar{p}$ to have the opposite value to that of p.

Pseudocode 5 Algorithm `PDRplan`$(X, s_I, g, \mathcal{A})$:

Input:

A positive STRIPS planning problem $\mathcal{P} = (X, s_I, g, \mathcal{A})$

Output:

A plan for $\mathcal{P}$ or a guarantee that no plan exists

$L_0 \leftarrow \{\{p\} \mid p \in g\}$ /* The goal cube treated as a set of unit clauses */
foreach $j > 0 : L_j \leftarrow \emptyset$
for $k = 0, 1, \ldots$ **do**
 /* Path construction: */
 if $s_I \models L_k$ **then**
  $Q \leftarrow \{(s_I, k)\}$
  while Q **not** empty **do**
   pop some (s, i) from Q with minimal i
   if $i = 0$ **then**
    return PLAN FOUND
   if `extend`(s, L_{i-1}) returns a successor state t **then**
    $Q \leftarrow Q \cup \{(t, i-1), (s, i)\}$
   else
    `extend` returned a reason $r \subseteq \mathit{Lits}(s)$
    foreach $0 \leq j \leq i : L_j \leftarrow L_j \cup \{\neg r\}$

    /* Obligation rescheduling: */
    if $i < k$ **then**
     $Q \leftarrow Q \cup \{(s, i+1)\}$

 /* Clause propagation: */
 for $i = 1, \ldots, k+1$ **do**
  foreach $c \in L_{i-1} \setminus L_i$ **do**
   /* Clause push check */
   $s_c \leftarrow \{p \mapsto \mathbf{0} \mid p \in c\} \cup \{p \mapsto \mathbf{1} \mid p \in (X \setminus c)\}$
   if for every $a \in \mathcal{A} :\ s_c \not\models \mathit{pre}_a$ or $\mathit{apply}(s_c, a) \not\models L_{i-1}$ **then**
    $L_i \leftarrow L_i \cup \{c\}$
  /* Convergence check */
  if $L_{i-1} = L_i$ **then**
   return NO PLAN POSSIBLE

A version of PDR specialized to positive STRIPS planning is shown in Pseudocode 5. The calls to a SAT-solver of the original formulation were replaced, respectively, by a simple entailment check (line 5), a call to the `extend` procedure (line 11), and by an enumeration of the successor states of the state s_c as defined in Lemma 4 (line 26).

4.4 Reversing the Search Direction

It has been mentioned that the original formulation of PDR is based on the opposite search direction than the one described in this paper and extends the paths from a goal state backwards towards the initial state. We would like to test the algorithm with both directions to see which one is more favorable in practice.

One possibility to achieve this is to provide PDR with an inverted version of the input, where the initial and goal states have been swapped and the transition relation "turned around". This is straightforward to do when the input is an STS (recall Section 2.3), as is the case with the general version of PDR. The situation is more complicated with the SAT-solver free version, which directly takes a STRIPS planning problem as an input. Indeed, it seems the `extend` procedure substantially relies on the forward direction.

Interestingly, there exists a transformation for inverting STRIPS planning problems. It was first described by Massey (1999) in his dissertation. We present here a more streamlined version due to Pettersson (2005) which relies on the problem being positive. Let us start by introducing an alternative representation of positive STRIPS planning problems, which makes the description of the transformation particularly straightforward. A positive STRIPS planning problem in the *subset representation* is given by a tuple $\mathcal{P} = (X, i, g, \mathcal{A})$, where $i, g \subseteq X$ are the initial and goal conditions, respectively, and every action $a \in \mathcal{A}$ is encoded by a triple $a = (pre_a, add_a, del_a)$, consisting of a precondition list, an add list and a delete list, which are subsets of X such that $pre_a \cap add_a = \emptyset$ and $add_a \cap del_a = \emptyset$. The subset representation differs from the one presented in Section 2 by encoding the initial state by the set of those variables that are true in it:

$$s_I(p) = 1 \text{ if and only if } p \in i,$$

and by splitting action's effects into positive and negative ones:

$$\mathit{eff}_a = add_a \cup \{\neg p \mid p \in del_a\}.$$

The goal condition g and precondition lists pre_a remain intact, but now may be understood as subsets of X since the problem is positive. It should be clear that the subset representation and the one of Section 2 are equivalent.

Now, for an action $a = (pre_a, add_a, del_a)$ an inverted action a^{-1} is formed by exchanging the precondition and delete list: $a^{-1} = (del_a, add_a, pre_a)$. For a set of actions $\mathcal{A}$ the set of inverted actions is $\mathcal{A}^{-1} = \{a^{-1} \mid a \in \mathcal{A}\}$. Given a planning problem $\mathcal{P} = (X, i, g, \mathcal{A})$ in the subset representation, the *inverted* problem $\mathcal{P}^{-1}$ is obtained by exchanging the initial and goal conditions while taking their complements with respect to X and using the inverted action set:

$$\mathcal{P}^{-1} = (X, (X \setminus g), (X \setminus i), \mathcal{A}^{-1}).$$

The original problem and its inverted version are related in the following sense:

Theorem 2. *The sequence of actions $a_0, a_1, \ldots, a_k$ is a plan for the planning problem $\mathcal{P}$ if and only if the sequence $a_k^{-1}, a_{k-1}^{-1}, \ldots, a_0^{-1}$ is a plan for $\mathcal{P}^{-1}$.*

This means that performing forward search (also called progression) in $\mathcal{P}$ is equivalent to performing backward search (regression) in $\mathcal{P}^{-1}$ and vice versa. Notice that, a priori, there is no computational overhead incurred by the transformation: the inverted problem has the same number of actions as well as the same set of state variables X and so the representation of states is of the same size. A proof of Theorem 2 along with further intuition behind the transformation and its theoretical and practical implications are described by Suda (2013b).

4.5 Further Improvements

We describe three additional modifications of PDR that aim to make the algorithm more efficient in practice. While the first is a planning-specific improvement of the `extend` procedure, the other two focus on how obligations are handled by the overall algorithm. In Section 5 we experimentally evaluate the effect of these modifications on solving planning problems. The interested reader can find the pseudocode of the modifications in Appendix A.

4.5.1 Lazy False Clause Computation

One way to speed up the `extend` procedure in practice is a technique we named *lazy false clause computation*. It is based on the following two observations:

- L^s, the set of clauses false in the state s, is typically only a small subset of L, the set of all the clauses the successor state should satisfy,
- only a small fraction of the available actions makes any of the clauses of L^s true in their respective successor.

The idea is to avoid the relatively expensive computation of the set of clauses false in the successor t, i.e. the computation of the set L^t on line 10 (Pseudocode 3), and instead first only look at the truth value of the clauses from L^s. (Notice that L^s is precomputed before we start iterating over the actions.) Only if we find an action a such that all its preconditions are satisfied in s and it makes all the clauses from L^s true in the respective successor t, we classify the action as *promising* and go back to computing the full L^t. Thus, with non-promising actions we save computational time. We may pay for it a little on the side of the quality of the reason set, because for them we only use $L^{s,t} = \{c \in L^s \mid t \not\models c\}$ instead of the full L^t for computing the reasons. On the other hand, with promising actions a complete test is necessary to distinguish a true successor t satisfying all of L from an action that "repairs" everything which was false in s, but "breaks" something else instead.

4.5.2 Sidestepping

Sidestepping is a technique we propose to make PDR more active in early exploration of promising paths. It partially circumvents the limitation stemming from the fact the `extend` procedure emulates the sequential encoding $\mathcal{S}_{\mathcal{P}}^{seq}$.

Imagine we want to extend an obligation (s, i), i.e. to find a successor of s that would satisfy L_{i-1}, and there are two clauses $c_1, c_2 \in L_{i-1}$ false in s. Let us think of the two clauses as of two independent subgoals to be achieved. There are two actions a_1 and a_2 applicable in s. Action a_1 makes c_1 true in the successor state and a_2 makes c_2 true, but no action can make both the clauses true in one step. This means the extension cannot be successful and PDR will learn a new clause $c = c_1 \vee c_2$ (or a superset thereof). The clause c expresses the fact that in order to reach a state satisfying L_{i-1} in one step, at least one of the two clauses c_1, c_2 must be satisfied beforehand. This could be an important ingredient to showing that no path of length k can reach the goal, helping the algorithm eventually advance to the next iteration. However, because in planning we are usually more interested in actually finding plans than showing their non-existence, deriving the clause c could represent unnecessary extra work. The idea behind sidestepping is to make the `extend` procedure succeed more often, even if that does not mean directly advancing into the next layer. In our example, we return the successor state $t = \mathit{apply}(s, a_1)$ with an additional flag informing the caller that the new obligation should have index i and not the usual $i-1$. We are effectively sidestepping from (s, i) to (t, i). In the next round the obligation (t, i) will be picked and successfully (provided the actions a_1 and a_2 do not interfere) extended into $(u, i-1)$ via the action a_2. This way we end up with a state satisfying L_{i-1} almost as if we executed the two actions a_1 and a_2 in parallel.

Let us now present the sidestepping technique in more detail. In order for an action a and its respective successor $t_a = \mathit{apply}(s, a)$ to qualify for a sidestep during extension of an obligation (s, i) the following conditions must to be met:

1) a is applicable in s,

2) t_a improves over s with respect to the set of satisfied L_{i-1} clauses:

$$L_{i-1}^{t_a} = \{c \in L_{i-1} \mid t_a \not\models c\} \subset L_{i-1}^{s} = \{c \in L_{i-1} \mid s \not\models c\},$$

3) t_a satisfies all the L_i clauses.

Notice that we require the improvement to be strict (condition 2). This ensures that sidestepping does not compromise termination. We also make sure that the new state stays within L_i (condition 3) – an improvement in one respect should not be payed for by an overall deterioration.

If there is no action that qualifies for a sidestep, we compute and return a reason set as usual. Otherwise, we choose among them an action a for which the size of $L_{i-1}^{t_a}$ is the smallest. The case when $|L_{i-1}^{t_a}| = 0$ corresponds to a regular successful extension and a new obligation $(t_a, i-1)$ will be stored in the set Q. If $|L_{i-1}^{t_a}| > 0$, we store (t_a, i) instead, which means that we perform a sidestep.

After sidestepping both the old obligation (s, i) and the new (t_a, i) occupy the same index in Q. It is important that we prioritize the latter over the former for picking (e.g., even if we otherwise want to use queue tie-breaking strategy; see Section 3.6.4), by which we prevent the algorithm from sidestepping in the same way more than once.[14]

Notice that sidestepping is an extension of PDR that relies on a modification of the planning-specific `extend` procedure. As such it does not have an immediate counterpart in the original algorithm, where path extensions are delegated to a SAT-solver.

14. By the time (s, i) is reconsidered we must have had an unsuccessful extension of (t_a, i), which means L_i got in the meantime strengthened and t_a no longer satisfies it.

4.5.3 Keeping Obligations between Iterations

Let us return to obligation rescheduling (lines 18 and 19 of Pseudocode 5) to discuss one additional aspect of this feature. Notice that we reschedule an obligation (s, i) only if $i < k$ so that the new obligation $(s, i + 1)$ is never positioned further from the goal than k steps during iteration k. Obligations of the form (s, k) are simply forgotten which ensures that the path construction phase eventually terminates. A viable alternative to this strategy is to reschedule these obligations on the queue Q with index $k+1$, but set them into a "dormant state" and return to them only during the next iteration. This can be understood as effectively enlarging the set of initial states for the next iteration so that it includes all the states reached so far.

Although this modification is quite simple to implement and seems to go well with the spirit of obligation rescheduling itself, it has not been publicly described yet. A potential disadvantage could be the increased memory consumption, since all the states ever encountered during the run must be stored by the algorithm. The utility of this modification may, therefore, depend on the application domain.

5. Experiments

In this section we report on a series of experiments aimed to establish the practical relevance of PDR for automated planning. We first compare the standard version of the algorithm combined with encodings to the SAT-solver free variant of PDR proposed in this paper. The latter is implemented in our new planner PDRplan. Next, we measure the influence of the several design choices mentioned in Section 3 as well as of the various improvements proposed in Section 4.5 on the performance of PDRplan. The most successful configuration of PDRplan is then compared to other planners, including state of the art representatives of the heuristic search planning and planning as satisfiability paradigms. Finally, we also assess PDRplan from the perspective of plan quality, finding optimal length plans and detecting unsatisfiable problems.

5.1 The Setup

We performed the experiments on machines with 3.16 GHz Intel Xeon CPU, 16 GB RAM, running Debian 6.0. Although multiple cores are available on each machine, all the planners used only one core and we made sure that there was no other busy process running concurrently that would compete with a planner for memory, etc. The main measured resource was computation time. We used a time limit of 180 seconds per problem instance for most of the runs, but increased it to 1800 seconds for the main comparison.

To increase the level of confidence towards the correctness of our implementation all the generated plans were subsequently checked by the latest version of plan validator `VAL` (Howey, Long, & Fox, 2004). No discrepancies were found during the experiments reported in this paper.

We tested the planners on the STRIPS[15] benchmarks of the International Planning Competition (IPC, 2014). So far, there were altogether seven repetitions of the competition happening biennially from 1998 to 2008 and once in 2011. Each time the planners competed

15. The richer ADL formalism is currently not supported by PDRplan.

over several benchmark domains of various planning scenarios. We used all the available STRIPS domains except the following:

- 1998-MOVIE, where it turned out to be technically difficult to validate the plans.[16] Note that the domain is, in fact, trivial to solve.
- 2000-SCHEDULE, which is originally an ADL domain. The competition archive contains also a STRIPS version, but accompanied by a note saying that this version later proved to be problematic and was dropped from the competition.
- 2002-ROVERS, the problems of which are included in the set 2006-ROVERS.
- 2002-SATELLITE and 2011-TIDYBOT, which make use of actions with negative preconditions, a feature not supported by our parser.

Altogether, we collected 1561 problems in 49 domains (see Table 3 on page 304 for a detailed list). The 2008 and 2011 competition benchmarks specify action costs. We modified the respective files to remove this feature, which is not supported by PDRplan.

We implemented PDR with the `extend` procedure as described in Section 4 in the PDRplan system. The code of PDRplan (approximately 2K lines of C++) is built on top of a PDDL parser and a grounder adopted from SatPlan 2006 (Kautz, Selman, & Hoffmann, 2006). We modified the parser to successfully process the large problems of the more recent IPC domains. The source code of PDRplan is publicly available on our web page (Suda, 2014), which also contains all the other material relevant for reproducing the experiments.

5.2 PDRplan v.s. Standard PDR plus Encodings

The main purpose of the first experiment was to compare PDRplan and its planning-specific implementation of the `extend` procedure to a composition of the general PDR, which uses a SAT-solver to answer the one-step reachability queries, with various encodings of planning into an STS. We also wanted to establish which of the two possible search directions in PDR is more favorable for discovering plans.

We took our implementation of PDR called `minireachIC3`, originally developed as a model checking tool for hardware circuits. It internally relies on the SAT-solver Minisat (Eén & Sörensson, 2003) version 2.2. We extended `minireachIC3` to be able to read a description of an STS. We designed a new input format for that purpose, which we call DIMSPEC (Suda, 2013a). It is a simple modification of the well-known DIMACS CNF format used by most SAT-solvers extended to define the individual clause sets of an STS.

We coupled `minireachIC3` with four encoders of planning into an STS. The first two encoders, `seq` and `par`, are our implementations of the two simple encoding $\mathcal{S}_{\mathcal{P}}^{seq}$ and $\mathcal{S}_{\mathcal{P}}^{par}$, respectively (recall Section 2.6). The third encoder is a version of the planner `Mp` (Rintanen, 2012) modified to output the encoded instance in the form of an STS and quit before starting the actual solving process. `Mp` uses the $\exists$-step parallel encoding scheme of Rintanen et al. (2006). Finally, the fourth encoder implements the SASE encoding scheme introduced by

16. The parser we adopted for PDRplan removes vacuous arguments of operators from the resulting actions' names. The validator `VAL` then complains about the resulting plans.

Huang et al. (2012). The particular implementation we used derives from the `FreeLunch` planning library (Balyo et al., 2012).

In order to obtain a fair comparison we used a basic version of PDRplan configured in a way that most resembles the workings of `minireachIC3`. The configuration follows the planning-specific version of the overall algorithm (Pseudocode 5) and relies on the `extend` procedure (Pseudocode 3) with the minimization phase of the reason computation (stage three) enhanced by induction (Pseudocode 4). The additional improvements of Section 4.5 were disabled for this experiment.

We compared the systems in both search directions. By the *forward* direction, we mean the one preferred in this paper, where PDR constructs the path from the initial state towards the goal. The opposite, *backward* direction is preferred by the original exposition of PDR used in model-checking. To start `minireachIC3` in the backward direction we inverted the encoded STS, to reverse the search direction of PDRplan we inverted the planning problem (as explained in Section 4.4).[17]

5.2.1 Adding Invariants

An invariant of a transition system is a property of the initial state preserved by all transitions. In planning, one typically considers invariants in the form of binary clauses (Rintanen, 1998), which can be computed by a simple fixpoint algorithm (Rintanen, 2008b). Adding the invariant clauses into an encoding is known to speed up plan search in the planning as satisfiability paradigm.

We noticed that the performance of PDRplan in the backward direction can also be enhanced with the help of invariants. When PDR is run in the backward direction, it is sound to strengthen every layer by the binary clauses of a precomputed invariant. These clauses then help to guide the path construction towards the initial state. Adding invariants in the forward direction does not make sense for PDRplan, because all the generated states are reachable from the initial state and, therefore, satisfy the invariant automatically.[18]

We used the same invariant generation algorithm as in PDRplan to also enhance the encodings `seq` and `par` for the run of `minireachIC3`. It turned out that in case `minireachIC3` invariants slightly help even in the forward direction.[19] We note that binary clause invariants are also explicitly included in the Mp encoding and implicitly present in the SASE encoding, which relies on the SAS^+ planning formalism (Bäckström & Nebel, 1995) to which a STRIPS problem is converted with the help of invariants (Helmert, 2009).

5.2.2 Detecting Auxiliary Transition Variables

It is essential for a good performance of `minireachIC3` combined with encodings that the algorithm does not make decisions prematurely.

17. One could also experiment with encodings of the inverted problems. We leave this for future work.
18. In theory, there is a corresponding notion of a backward invariant, a property of the goal states preserved by traversing the transitions backwards. Symmetrically, backward invariants could be used to enhance the performance of forward PDR. In practice, however, while standard invariants are typically very useful, there is rarely a non-trivial binary clause backward invariant in the planning benchmarks.
19. This can be explained by observing that the SAT-solver does not necessarily construct the successor state by first choosing an action (or a set of actions, in the case of `par`), which would then fully determine the successor. When it starts by deciding on the state variables of the successor, invariants become useful.

Example 4. Consider a run of the algorithm in the forward direction with the encoding $\mathcal{S}^{seq}_{\mathcal{P}}$. Because in this encoding the action variables A occur in the unprimed part of the transition clauses T^{seq}, any given state s, being an assignment over $\Sigma = X \cup A$, already stores the information about the action that will be applied next and, therefore, fully determines the value of the state variables X' of its successor. As a result, contrary to the intuition, the evaluation of the extension query

$$SAT?[\,Lits(s) \wedge T^{seq} \wedge (L)'\,]$$

does *not* boil down to choosing an action applicable in the state $(s \restriction X)$ of the original planning task, such that the successor state would satisfy the clauses of L, but instead involves choosing an action to be applied in the already determined successor such that the successor (as a valuation over X') and the chosen action (as a valuation over A') together satisfy the clauses from $(L)'$, which, in general, span the whole signature Σ'. We can see that, in some sense, all the decisions are made one step too early.

We observed a marked improvement in the performance of `minireachIC3` combined with encodings when we extended the tool with a preprocessing step that detects *auxiliary transition variables* in the unprimed part of the transition clauses and re-encodes them into the primed part in order to avoid committing to decisions prematurely as demonstrated in the example above. Formally, given an STS $\mathcal{S} = (\Sigma, I, G, T)$, auxiliary transition variables Aux are those variables of Σ that do not appear in I or G and, when primed, are not shared by T and T'. This means that

$$Aux' = \Sigma' \setminus (\,Vars(I') \cup Vars(G') \cup (\,Vars(T) \cap Vars(T'))).$$

The action variables A of the $\mathcal{S}^{seq}_{\mathcal{P}}$ encoding are an example of auxiliary transition variables. For every transition clause $c \in T$, the preprocessing step identifies literals $l \in c$ such that $Vars(l) \in Aux$ and turns each such l into l'. The soundness of the transformation is easy to establish.

5.2.3 Results of the Experiment

The results of the first experiment can be found in Figure 3. There are several observations to be made. As foretold, the forward direction is generally more successful than the backward. Within the time limit of 180 seconds, each of the five systems solves more problems in the forward direction than in the backward direction. We see that in the backward direction, invariants help to improve the performance of PDRplan. Nevertheless, within that direction `minireachIC3` combined with the Mp encoding is more successful. The most successful system is PDRplan in the forward direction. It solves 8.6 percent more problems than the second best system, `minireachIC3` combined with the Mp encoding in the forward direction. Although we do not consider these results as a definitive answer to the "PDRplan vs. encodings" question,[20] we decided to only focus on PDRplan in the forward direction for (most of) the subsequent experiments.

The overall trends captured by Figure 3 are most of the time respected when the comparison is performed on level of individual problem domains (comparing the number of

20. For instance, replacing Minisat in `minireachIC3` by a more recent and more efficient SAT-solver could change the picture to a certain degree.

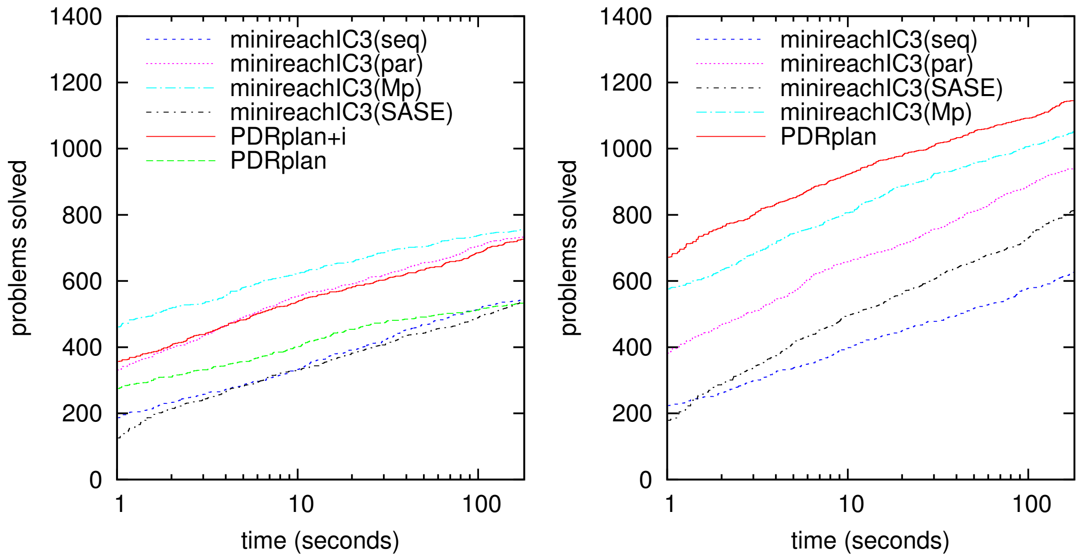

Figure 3: Comparing PDRplan to `minireachIC3` combined with encodings. Number of problems solved within the given time limit is shown, separately for the backward direction (left) and the forward direction (right).

problems solved in 180 seconds), nevertheless there are some notable exceptions worth mentioning.

- On the LOGISTICS domain PDRplan behaves better in the backward direction and without invariants. The best system on this domain, however, is `minireachIC3` with Mp encoding in the forward direction.
- The relatively difficult 2011-BARMAN domain is almost fully solved (19 out of 20 problems) by PDRplan in the backward direction with invariants. The second best system on this domain is `minireachIC3` with `par` encoding in the backward direction with only 7 problems solved.
- The following are among the domains where PDRplan is not the best system: 1998-MYSTERY (`minireachIC3` with SASE and Mp encodings in the forward direction both solve 5 problems more), 2004-PHILOSOPHERS (18 more problems solved by `minireachIC3` both with `par` and Mp encodings in the forward direction), and 2011-VISITALL (`minireachIC3` with Mp encoding solves 4 more problems).
- There are several domains where `minireachIC3` with Mp encoding is better in the backward direction than in the forward direction. The difference is most pronounced on 2006-OPENSTACKS, and 2011-FLOORTILE.

The observation of the last point is in accord with how the Mp encoding is used with the Mp planner itself. What Rintanen (2012) describes is effectively a depth-first backward chaining planning algorithm inside the SAT-solving framework. This can be seen to be very

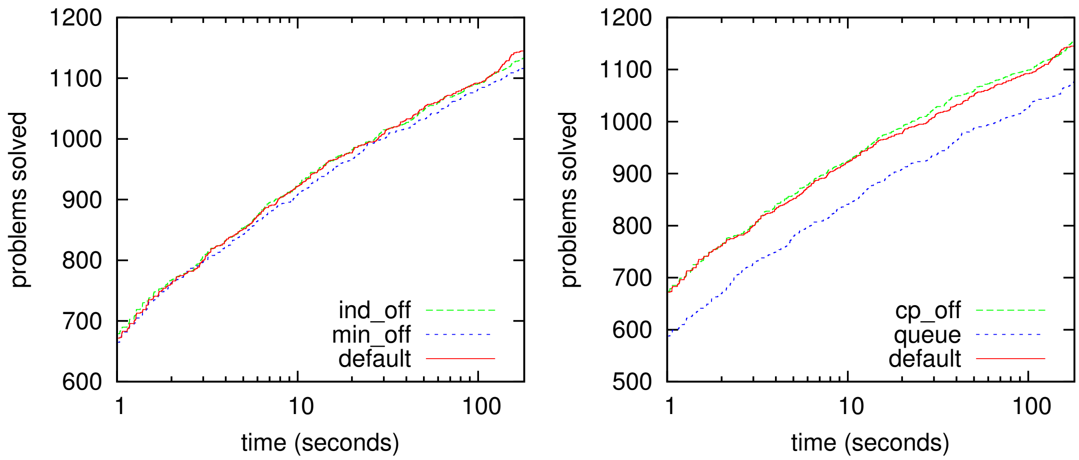

Figure 4: Tuning PDRplan. The effect of explicit (inductive) reason minimization (left), and clause propagation and the queue tie-breaking strategy (right).

close to backward PDR when coupled with the same encoding. We hypothesize that the suitability of the Mp encoding for the backward direction of search emerges also with PDR.

5.3 Tuning PDRplan

In the second experiment (see Figure 4) we focused on several features of the standard PDR and tried to established their importance for solving planning problems. We used PDRplan in the forward direction and 180 seconds time limit. We measured the effect of each feature separately with the reference configuration denoted as default. This is the same configuration as the one used in the previous experiment.

5.3.1 Explicit (Inductive) Reason Minimization

By explicit minimization we mean the optional stage three of the reason computation in the `extend` procedure, which can be enhanced by induction as described in Section 4.2. In Figure 4 (left) we compare the performance of the default configuration, which relies on the inductive version of reason minimization (Pseudocode 4), to configuration ind_off, which does not use induction and implements minimization as described in Pseudocode 3, and to configuration min_off, which skips the optional stage three altogether.

We can see that while the positive effect of explicit minimization is slight but consistent along the time axis, induction only starts to pay off on the global scale when the time limit exceeds 100 seconds. At the 180 seconds mark the ind_off configuration solved 1.0 percent fewer and the min_off configuration 2.4 percent fewer problems than default.

Per domain view reveals that induction is especially important for the success on 2000-BLOCKS, 2004-PHILOSOPHERS, and 2008-CYBER-SECURITY. On domains such as 2002-ZENOTRAVEL or 2008-TRANSPORT it is better to turn minimization off completely, and there are also domains, such as 1998-MYSTERY or 2006-TRUCKS, where it pays off to minimize, but not inductively. In the last two categories, however, the difference is never by more than a problem or two per domain and thus it could be potentially equalized within a higher time limit.

Interestingly, out of the total of 1561 problems, the execution of default and ind_off diverged only on 145 problems.[21] This means that on most of the problems induction does not help to minimize reasons beyond what can be achieved with non-inductive minimization. To give another statistics, we note that over the whole problem set during a call to the `extend` procedure inductive minimization removes 1.49 literals and computes a reason with 50.60 literals on average. The non-inductive minimization in min_off removes 1.44 literals and generates a reason with 51.22 literals on average.

5.3.2 Clause Propagation

In Figure 4 (right) we can compare the default configuration to a configuration in which clause propagation has been turned off (cp_off). We see that clause propagation slows PDRplan slightly down without any clear benefit before the 180 seconds mark. Although a later independent experiment with a 1800 second time limit showed that clause propagation can be useful on planning problems, it is questionable whether the effect justifies the relatively high effort connected with implementing the technique.[22]

A closer look reveals that only on 28 percent of the tested problems a clause was successfully pushed forward during the 180 seconds bounded runs. This may seem to be in contrast with the experience from hardware model checking where clause propagation plays a key role. Its main effect there, however, lies in speeding up convergence of PDR on the unsatisfiable problems. Since more than 99 percent of our planning benchmarks are satisfiable, this role of clause propagation cannot be demonstrated. In fact, we also independently confirmed in an experiment with `minireachIC3` on *satisfiable* hardware benchmarks[23] that clause propagation is not beneficial in the *forward* direction.

5.3.3 Stack vs. Queue Tie-breaking

Here we evaluate the effect of the strategy for breaking ties during popping obligations from the set Q (see Section 3.6.4). The stack strategy used in the default configuration is compared to the curve of the queue strategy in Figure 4 (right). The queue strategy solves about 5.9 percent fewer problems in total. However, there are 18 problems solved by the queue strategy only (and 85 problems solved only by the stack strategy). The most interesting observations on the per domain scope are probably

- 59 problems (out of 60) from the 2000-BLOCKS domain solved by the stack strategy compared to only 36 solved by the queue strategy, and
- 2 problems (out of 20) from the 2011-BARMAN domain solved by the queue strategy compared to 0 problems solved by the stack strategy.

21. By either generating a different number of obligations before a successful termination or differing in whether they successfully terminated at all before the 180 seconds mark.
22. In the final comparison to other planners (see Section 5.5) clause propagation is responsible for 6 additional problems scored by PDRplan.
23. On benchmarks from the Hardware Model Checking Competitions 2007–2012 (Biere, Heljanko, Seidl, & Wieringa, 2012) with a time limit of 100 seconds.

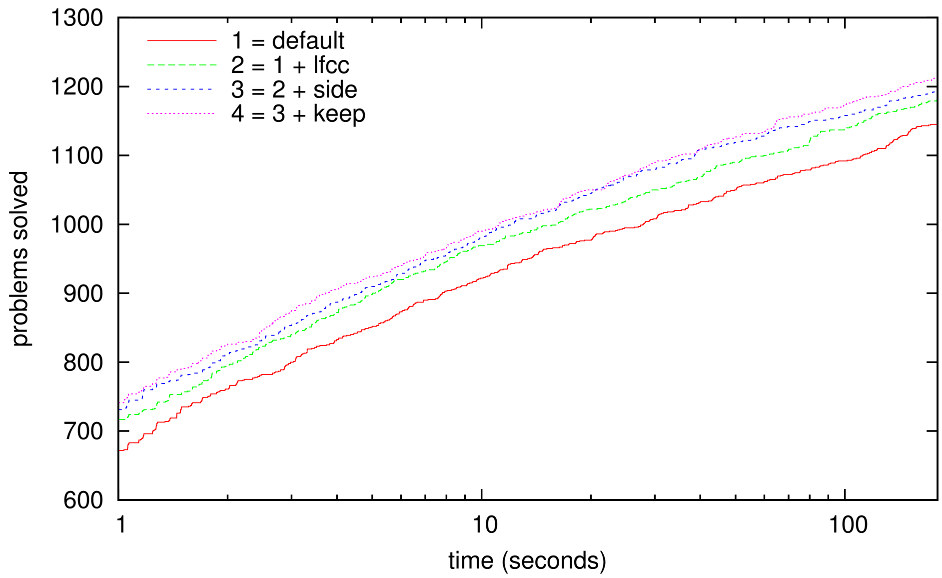

Figure 5: Improving PDRplan. The default configuration is progressively extended by turning on three different techniques.

Preferring to explore longer paths before short ones has the unpleasant side effect that also the plans discovered by the stack strategy tend to be longer. Measured over the 1055 problems solved by both strategies, the plans generated by the stack strategy are on average 24 percent longer.

A more detailed discussion on the topic of plan quality is postponed till Section 5.6.

5.4 Improving PDRplan

The purpose of the third experiment was to evaluate the three improvements proposed in Section 4.5. These were successively: 1) lazy false clause computation (lfcc), 2) the sidestepping technique (side), and 3) keeping obligations between iterations (keep). Figure 5 displays the effect of progressively enabling the three techniques in the presented order. We see that to varying degrees each technique represents an improvement and each successive configuration solves more problems.

A different perspective is provided by Table 2 which also reveals how many problems were uniquely solved by only one of the two successive configurations. It shows that none of the improvements are unambiguous across the whole problem set and that there are exceptions to the prevailing trends.

These can be best highlighted on the level of individual domains. For instance, the number of solved problems drops on 2000-BLOCKS and 2008-CYBER-SECURITY with lazy false clause computation (configuration 2), but it is improved again by the subsequently enabled techniques. Sidestepping (configuration 3) makes the performance worse on 2002-DRIVERLOG, 2004-SATELLITE, or 2008-CYBER-SECURITY. On the other hand, the technique represents a huge improvement on the 2004-OPTICAL-TELEGRAPH domain

configuration	total	delta	gained	lost
1 = default	1145	–	–	–
2 = 1 + lfcc	1180	35	55	20
3 = 2 + side	1195	15	67	52
4 = 3 + keep	1212	17	42	25

Table 2: Number of problems solved within 180 seconds (total). The difference (delta) between two successive configurations decomposed into additionally solved problems (gained) and problems only solved without the improvement (lost).

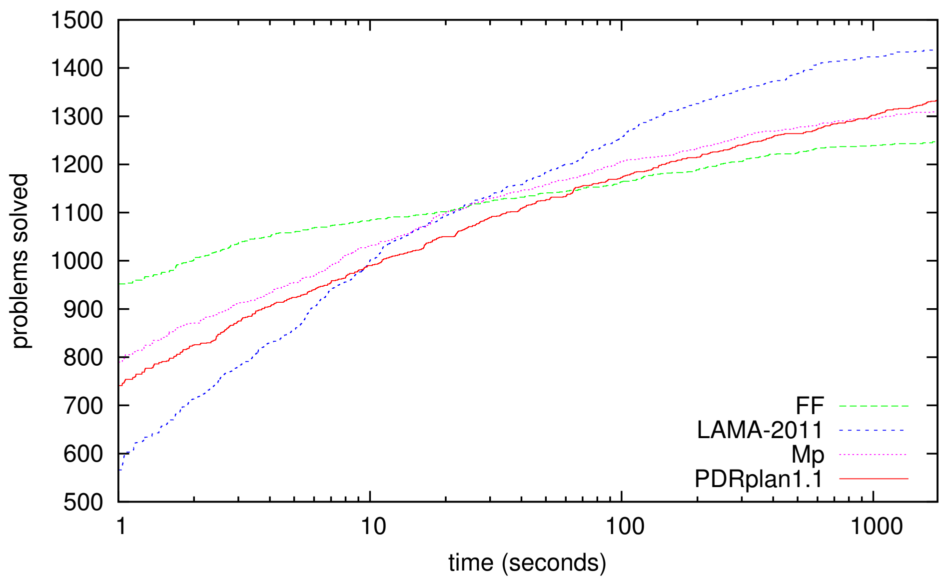

Figure 6: Comparing the final version of PDRplan to other planners. Showing the number of problems solved within the given time limit.

(from 2 to all 14 problems solved) and on 2004-PHILOSOPHERS (from 11 to all 29 problems solved). Finally, keeping the obligations (configuration 4) is detrimental to the performance on the 2011-FLOORTILE domain (the number of solved problems drops from 19 to 13), but the technique, for example, helps to "recover" the 2008-CYBER-SECURITY problems that were "lost" due to sidestepping.

5.5 Comparing to other Planners

We compared the improved PDRplan – the configuration 4 from the previous experiment denoted here PDRplan1.1 – to the following planners:

- The planner FF (Hoffmann & Nebel, 2001) as a baseline representative of heuristic search (Bonet & Geffner, 2001) planners. We used version 2.3, but enhanced its input module to make it cope with the large problems of the more recent IPC domains. The default parameters for FF have been used.

- The planner Fast Downward (Helmert, 2006), the current state of the art heuristic search planner. We used the configuration LAMA-2011 (Richter & Westphal, 2010), the winner of the satisficing track of IPC 2011.

- The Mp planner (Rintanen, 2012), probably the current best representative of the planning as satisfiability approach (Kautz & Selman, 1996). We used version 0.99999 with default parameters.

For this experiment the time limit was increased to 1800 seconds.

The overall performance of the planners can be seen in Figure 6. The planner FF has a very fast startup and solves the most problems (952) within one second. However, FF is the worst of the planners to make use of the additional time and solves the fewest problems (1247) in total. On the opposite side stands LAMA-2011 with the slowest startup (566 problems within one second), but with the best total (1437). PDRplan1.1 and Mp are close to each other in performance both at the beginning – PDRplan1.1 solves 741 and Mp 790 problems within one second – and at the end – in total PDRplan1.1 solves 1333 problems gaining a slight edge over Mp with 1310 problems solved.

Table 3 shows a domain-by-domain decomposition of the results. We can see that there are several domains where PDRplan1.1 solved the most problems of the four planners, namely the 2000-BLOCKS, 2002-FREECELL, 2004-PIPESWORLD-NOTANKAGE, and 2006-TRUCKS domains. The domains 2004-PHILOSOPHERS, 2006-PATHWAYS, and 2006-STORAGE were completely solved by only PDRplan1.1 and Mp. On the hand, a comparatively poor performance of PDRplan1.1 can be observed on the 1998-LOGISTICS and 1998-MPRIME domains, and also on the 2011-PARKING (shared with FF) and 2008+2011-SOKOBAN (shared with Mp) domains.

5.6 Plan Quality

IPC 2008 (Helmert, Do, & Refanidis, 2008) introduced a criterion for measuring planner performance which takes into account the quality of the obtained plans. For every problem solved, a planner aggregates a *score* computed as the ratio c^*/c, where c is the cost[24] of the returned plan and c^* the cost of the best known plan (either a plan computed beforehand by the competition organizers or the best plan found by any of the participating systems). When viewing the results of the previous experiment through the lenses of this criterion, one discovers that PDRplan1.1 drops from the second place to the last.

We reviewed all the previously discussed features and improvements and discovered that the configuration of PDRplan1.1 is not the best possible with respect to plan quality. In particular, by switching to the queue tie-breaking strategy (we denote the respective configuration PDRplan1.1+queue) the aggregated score of the planner improves. A slight

24. As mentioned before, we do not consider action costs in this paper, so, in our setting, a cost of a plan is simply equal to its length.

	size	PDRplan1.1	FF	LAMA-2011	Mp
1998-GRID	5	5	5	5	5
1998-GRIPPER	20	20	20	20	20
1998-LOGISTICS	35	18	35	35	32
1998-MPRIME	35	25	34	35	34
1998-MYSTERY	30	19	18	23	19
2000-BLOCKS	60	**60**	48	55	46
2000-ELEVATOR	150	150	150	150	150
2000-LOGISTICS	36	36	36	36	36
2000-FREECELL	60	57	60	60	44
2002-DEPOTS	22	21	21	22	22
2002-DRIVERLOG	20	18	18	20	20
2002-ZENOTRAVEL	19	19	19	19	19
2002-FREECELL	20	**20**	19	19	15
2004-AIRPORT	50	40	38	33	49
2004-PIPESWORLD-NOTANKAGE	50	**45**	32	44	42
2004-PIPESWORLD-TANKAGE	50	37	17	42	38
2004-OPTICAL-TELEGRAPH	14	14	14	14	14
2004-PHILOSOPHERS	29	**29**	14	13	**29**
2004-PSR	50	50	42	50	50
2004-SATELLITE	36	28	34	36	35
2006-OPENSTACKS	30	30	30	30	19
2006-PATHWAYS	30	**30**	20	29	**30**
2006-PIPESWORLD	50	32	21	40	25
2006-ROVERS	40	39	40	40	40
2006-STORAGE	30	**30**	18	19	**30**
2006-TPP	30	30	28	30	30
2006-TRUCKS	30	**27**	12	15	19
2008-CYBER-SECURITY	30	30	4	30	30
2011-BARMAN	20	6	0	20	8
2008+2011-ELEVATORS	50	40	50	50	50
2011-FLOORTILE	20	14	10	6	20
2011-NOMYSTERY	20	14	7	10	19
2008+2011-OPENSTACKS	50	49	50	50	18
2008+2011-PARCPRINTER	50	50	50	50	50
2011-PARKING	20	8	7	20	20
2008+2011-PEGSOL	50	50	50	50	50
2008+2011-SCANALYZER	50	46	44	50	48
2008+2011-SOKOBAN	50	11	40	48	9
2008+2011-TRANSPORT	50	27	38	49	26
2011-VISITALL	20	9	4	20	0
2008+2011-WOODWORKING	50	50	50	50	50
TOTAL	1561	1333	1247	1437	1310

Table 3: Number of problems solved within 1800 seconds, grouped by domain. We highlighted those entries where PDRplan1.1 solves the most problems or shares the first place with one other planner. To save space the entries of IPC 2008 domains recurring later in IPC 2011 were merged with the respective entries of IPC 2011.

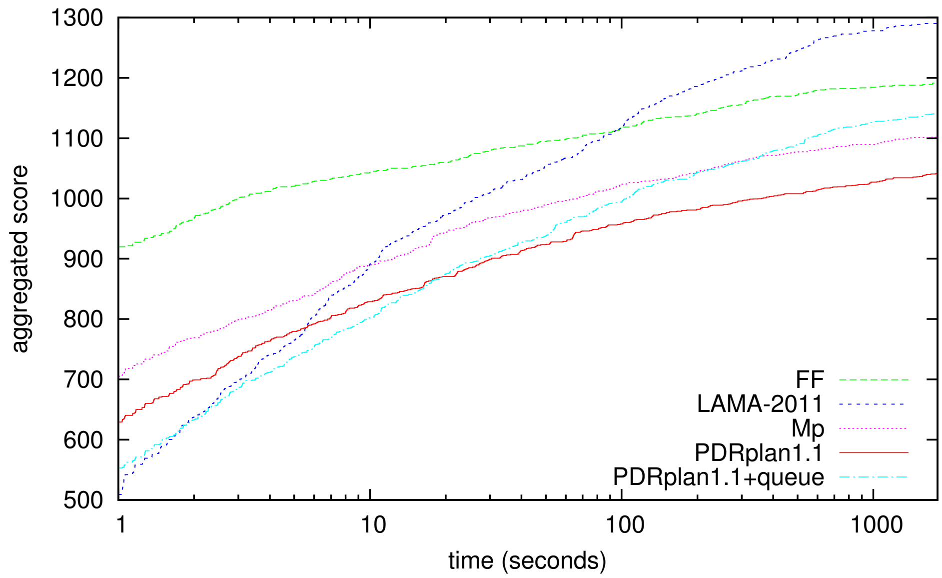

Figure 7: Comparing the planners with respect to plan quality. Showing the score aggregated by each planner within the given time limit.

improvement can also be observed when the lazy false clause computation is turned off in PDRplan1.1. Interestingly, doing both changes at once does not bring a combined benefit.[25]

Figure 7 shows the aggregated scores for the runs of the previous experiment together with a run of PDRplan1.1+queue.[26] Although PDRplan1.1+queue solves only 1263 problems in 1800 seconds (compared to 1333 solved by PDRplan1.1), it aggregates a score of 1141.1 points while PDRplan1.1 only reaches 1041.4. This means the former configuration catches up with Mp, which aggregates 1102.7 points.

We note that these statistics should be taken with a grain of salt, because they only provide a "plan quality view" on the satisficing runs of the planners. None of the systems was explicitly attempting to find short plans or making use of the fact that the time limit is 1800 seconds. Moreover, even in such a setting the plan quality can typically be improved afterwards by a post-processing of the discovered plans (Balyo & Chrpa, 2014). We later incorporated the polynomial Action Elimination algorithm (Nakhost & Müller, 2010) as a plan post-processor into PDRplan1.1 and we were able to improve its aggregated score by 7.0 percent. A more thorough investigation of the quality of plans produced by PDRplan, as well as by PDR in general, is left for future work.

25. It seems that the already "carefully advancing" PDRplan1.1+queue benefits from the speed provided by lazy false clause computation, whereas with the stack strategy it helps to wait for the more precise reasons (having lfcc turned off) that will not allow the planner to search too deep too often.
26. The reference values for the best known cost c^* were collected just from the runs in the figure.

5.7 Anytime PDR and Optimal Planning

Recall that PDR can be adjusted to perform optimal planning by turning off the obligation rescheduling technique (and sidestepping).[27] Alternatively, we can modify PDR to continue the computation after a first plan is found, but afterwards only reschedule obligations that can be part of an improving plan.[28] This "anytime version" of PDR progressively reports on better and better solutions until finally terminating with a guarantee that the last reported plan is an optimal one. This happens when it reaches an iteration i equaling the length of the best discovered plan.

In this experiment we focused on optimal planning with respect to the sequential plan semantics.[29] We compared the performance of the anytime version of PDRplan1.1 (counting only solutions provably shown to be optimal) to BJOLP (Domshlak et al., 2011), an optimizing version of Fast Downward, and to an optimizing configuration of Mp.[30] Ordering the planners by the number of problems optimally solved within 1800 seconds we obtain:

1. BJOLP with 668 problems solved,
2. PDRplan1.1-anytime with 360 problems solved, and
3. optimizing Mp with 325 problems solved.

This order is preserved on the level of individual domains, except for several domains where Mp does not end up last. Mp solves optimally the most problems from 1998-MYSTERY, 2000-BLOCKS, 2008-CYBER-SECURITY, and also from the 1998-MPRIME domain. The margin is exceptionally pronounced on the last domain, where Mp solves 32 out of 35 problems, while BJOLP solves 21 and PDRplan1.1-anytime only 20 problems.

5.8 Detecting Unsatisfiable Problems

Although the main focus of the planning community, as reflected by the International Planning Competition, has traditionally been on satisfiable problems only, more recently, the importance of detecting unsatisfiable instances is getting recognized and addressed (Bäckström, Jonsson, & Ståhlberg, 2013; Hoffmann, Kissmann, & Álvaro Torralba, 2014). According to the experience from hardware model checking, PDR should be particularly strong at detecting unsatisfiable instances. In our last experiment, we tried to established whether this also holds in planning.

As the test problems, we used a collection by Hoffmann et al. (2014) consisting of 8 domains and a total of 183 unsatisfiable benchmarks. Table 4 shows domain-by-domain coverage results (for a time limit of 1800 seconds) of the following configurations of PDR:

27. PDR then looks for minimal length witnessing paths with respect to the encoded transition relation T. Using an encoding with sequential plan semantics (as implicitly done by PDRplan) ensures optimizing the number of actions of the resulting plan.
28. Formally, we keep an obligation (s, i) if the length of the path from the initial state s_I to s plus the value of the index i does not exceed the length of the best plan found so far.
29. This choice ruled out systems like SatPlan (Kautz et al., 2006) or SASE (Huang et al., 2012) from the comparison, because they only optimize with respect to the parallel plan semantics.
30. It uses a sequential encoding (option `-P 0`), does not skip any horizon length (option `-S 1`), and evaluates a single horizon length at a time (option `-A 1`).

	size	PDRplan		minireachIC3 with par					M&S	
		fwd	bwd	fwd	bwd	noind	nocp	blind	cf1	cf2
3UNSAT	25	10	10	11	**15**	11	5	**15**	**15**	**15**
Bottleneck	30	19	24	**25**	23	20	22	10	10	21
Mystery	9	4	**9**	**9**	**9**	**9**	**9**	2	**9**	6
UnsNoMystery	25	12	11	3	13	13	6	0	**25**	**25**
UnsPegsol	24	14	8	14	8	8	4	**24**	**24**	**24**
UnsRovers	25	11	11	11	**20**	15	12	0	17	9
UnsTiles	20	0	0	0	0	0	0	**10**	**10**	**10**
UnsTPP	25	5	6	4	6	3	3	5	**9**	**9**
Total	183	75	79	66	94	79	61	66	119	119

Table 4: Unsatisfiable benchmarks. Number of problems solved within 1800 seconds, grouped by domain. Best scores per domain are typeset in bold.

- PDRplan, in the same configuration as in the first experiment (Section 5.2), both in the forward (`fwd`) and backward (`bwd`) direction.
- `minireachIC3` combined with the `par` encoding (with invariants), also in both directions (`fwd`, `bwd`), and, in the backward direction, also with the inductive minimization replaced by the non-inductive version (`noind`), and, independently, with the clause propagation turned off (`nocp`).

In addition, Table 4 also contains entries adopted from the work of Hoffmann et al. (2014). These belong to the Fast Downward planner equipped with three different heuristics:

- Configuration `blind` uses a heuristic which returns 0 on goal states and 1 elsewhere – it essentially proves unsatisfiability by enumerating all the reachable states.
- Configuration `cf1` and `cf2` each use a version of a *merge-and-shrink* (M&S) heuristic (Helmert, Haslum, & Hoffmann, 2007), specifically adapted for detecting unsatisfiable problems (Hoffmann et al., 2014). These were the two best performing configurations in the experiment of Hoffmann et al. (2014).

We note that Hoffmann et al. (2014) also used a time limit of 1800 seconds, but ran their experiment on 2.20 GHz Intel E5-2660 machines with a 4 GB memory limit. This means that the last three configurations could be potentially solve more problems in our setup.

5.8.1 Results of the Experiment

When comparing the various configurations of PDR, we can see that the backward direction is generally more successful than the forward, although not consistently across all the domains. Interestingly, `minireachIC3` with the `par` encoding in the backward direction solves more problems than PDRplan. In fact, a preliminary test with a lower time limit showed that on these benchmarks this configuration is the strongest of all those considered in our first experiment (Section 5.2). Finally, we can also see that both induction and clause propagation are consistently helpful for solving unsatisfiable problems.

PDR does not come out as a winner from the comparison to the heuristic approach of Hoffmann et al. (2014), although it is able to solve the most problems on four domains. On two other domains, however, PDR is even dominated by blind search, i.e. by a simple state space enumeration. It seems that more benchmarks will be needed to establish which of the two approaches is generally more successful at detecting unsatisfiable planning problems.

5.8.2 Performance on UnsTiles

PDR is particularly bad at "enumerating states" when there is little possibility to generalize from the encountered ones. This is manifested most clearly on the UnsTiles domain, from which PDR could not solve a single problem within the given time limit. The domain represents the well known sliding puzzle and contains 10 problems with 8 tiles in a 3×3 grid and 10 problems with 11 tiles in a 3×4 rectangular grid.[31] We ran PDRplan in the forward direction to the end on the one of the smaller, 3×3 instances. It took about a day to complete, processed 701704 obligations and terminated when all the clauses from layer 11, in total 181440 clauses, where pushed to layer 12 during the clause propagation phase of iteration 11.

Notice that $181440 = 9!/2$ is half the size of the state space. By the classical result of Johnson and Story (1879), the state space of the sliding puzzle decomposes into exactly two connected components depending on the value of a certain *parity* function defined on the states. Unsatisfiable instances are those where the parity of the initial state and the goal state are different. Because the state space consists of just of two components, on a unsatisfiable instance PDR must converge (with the repeating layer) to a CNF description of the component containing the goal state. As we can see, this description is as large (in the number of clauses) as the component itself (in the number of states), and thus on the sliding puzzle PDR does not benefit at all from the symbolic representation via CNF.

5.9 Summary

Let us summarize the empirical findings obtained in this section. We state them as general claims while keeping in mind that they are, in fact, derived from the performance on two particular benchmark sets: the main set of 1561 mostly satisfiable IPC problems and the set of 183 unsatisfiable problems used in the last experiment.

- When planning with PDR it pays off to look for a plan from the initial state towards the goal and not vice versa. In other words, progression is preferable to regression in PDR. This holds even when invariants are employed, which help to improve the performance of regression considerably.

  Unsatisfiable instances, however, are typically better detected via regression.

- On satisfiable problems the SAT-solver free variant of PDR with planning-specific `extend` procedure (as described in Section 4) is generally more successful than the standard version of the algorithm combined with various encodings.

31. Most famous is the 15 tiles puzzle on a 4×4 grid (Wikipedia, 2014).

- Neither clause propagation nor inductive minimization, two techniques which are normally deemed essential for the performance of PDR, are very helpful on satisfiable planning problems. The techniques are, however, useful for detecting unsatisfiability.

- There are various ways of tuning PDR and improving its performance for planning. We tried to identify a configuration of the algorithm that would be most successful in our setup and later used it in PDRplan for a comparison with other planners. For all the techniques that turned out to be an improvement on average, however, there were exceptions in the form of individual problems or domains were performance degraded. These represent an interesting opportunity for future investigations (see Section 7).[32]

- When compared to other planners PDRplan shows respectable performance. In fact, its performance is comparable to or even slightly better than that of the planner Mp, a state of the art representative of the planning as satisfiability approach. It also solves the most problems of all the tested planners on several domains. Although PDRplan does not reach the score of LAMA-2011, the presented results are quite encouraging, especially given that PDR is a relatively young algorithm with a potential for further improvements.

- When plan quality is more important than just the number of problems solved, it pays off to switch from the stack to the queue tie-breaking strategy in PDR. Such a configuration is then able to keep up with and improve upon the performance of Mp with respect to the plan quality metric based on aggregated score.

  Another option for improving plan quality is to employ a post-processing step which attempts to remove redundant actions from the generated plan (Balyo & Chrpa, 2014).

- PDR can be easily modified to look for increasingly better solutions when given sufficient time and to eventually terminate with an optimality guarantee (with respect to plan length). Although LAMA-2011 is much more successful in finding optimal plans, the fact the PDRplan's "natural" encoding follows the sequential plan semantics could be the reason why PDRplan scores higher than Mp in this respect.

6. Related Work: Graphplan

We have shown in this paper that PDR is an algorithm closely related to the planning as satisfiability approach, although with the planning-specific implementation of the `extend` procedure no explicit encoding is present. We also highlighted the connection to heuristic search planning, with the direct correspondence on the side of explicitly explored reachable states and a little more subtle one on the side of the guiding layers, which can be seen as a continually refined admissible heuristic estimator. What we would like to discuss here is a perhaps surprising relation of PDR to the well-known Graphplan planning algorithm by Blum and Furst (1997).

32. On the one hand, by looking at the problems where a particular technique leads to a poor performance, we can identify its weak points and attempt to improve the technique. On the other hand, instead of relying on an overall best configuration we can also try to decide prior to running the algorithm itself on a promising set of enabled features for a given problem based on the problem's characteristics.

The main data structure of Graphplan is a *planning graph*, a layered structure for compressed representation of reachability information about the given problem. The individual layers of the graph over-approximate the set of states reachable by a given number of sets of parallel actions and are computed incrementally by propagation of so called *exclusion relations* between actions and state variables. The planning graph is searched for a plan by a backward-chaining strategy, starting from the goal set and regressing it, in the sense of the parallel plan semantics, to subgoals that do not violate the exclusions of the respective layer. Candidate (sub)goal sets shown not to lead to a plan within a specific number of steps are *memoized* to avoid repeating the same work in the future.

As already shown by Rintanen (2008a) the exclusion relations of the planning graph are equivalent to binary clause representation of k-step reachability information. This means they could be represented inside PDR as binary clauses in the respective layers. We claim additionally that also the memoized goal sets could be stored as layer clauses at the respective position: the clause being simply the negation of the conjunctive description of the goal set. With these two observations in mind, we can state that

> Graphplan is essentially a version of PDR with a specific implementation of the `extend` procedure based on the parallel plan semantics.

This correspondence allows us to highlight some other differences between the two algorithms beyond the preferred semantics of the emulated encoding.

- While the planning graph is built systematically by Graphplan and search for a plan is only started (resumed) when a full new layer has been computed, in PDR the layer construction is lazy, being triggered by unsuccessful path extensions.

  Goal set memoization in Graphplan, however, follows the same lazy pattern.

- Graphplan does not attempt to reduce the size of a memoized goal set, so, apart from the binary clauses, it only deals with long clauses representing the negation of the goal set. Notice that this would in PDR correspond to returning the full reason set $\mathit{Lits}(s)$ after an unsuccessful extension of the state s.

  A *subset* memoization has later been proposed by Long and Fox (1999), which corresponds to finding smaller reason sets.

- Graphplan searches for a plan in the backward direction. In PDR, the direction can be changed, but forward is more successful.[33]

- There is no equivalent to obligation rescheduling in Graphplan and so the algorithm always searches for optimal plans (with respect to the parallel plan semantics).

  The wavefront heuristic described by Long and Fox (1999) in their enhancement of Graphplan, however, seems to overcome this limitation, similarly to rescheduling.

The realization that PDR is related to Graphplan made us curious about the differences of the two algorithms in practice. We set up a small experiment where we compared PDR

33. Changing the search direction in Graphplan by running in on an inverted problem (see Section 4.4) is possible, but would likely lead to fewer problems solved. This is related to the already mentioned observation that there are very few problems with non-trivial backward invariants in the benchmark set.

to a mature implementation of Graphplan within the planner IPP (Koehler, 1999). In order to bring PDR as close as possible to what Graphplan does, we represented it by `minireachIC3` combined with the simple parallel encoding $\mathcal{S}_{\mathcal{P}}^{par}$ (see Section 2.6) enhanced by the binary clause invariant (as explained in Section 5.2). We ran `minireachIC3` in the backward direction and with obligation rescheduling turned off, so, similarly to IPP, it was looking for optimal plans. When measuring the number of problems solved (out of the main problem set described in Section 5) within 180 seconds, we obtained 466 solved by IPP and 484 by our configuration of `minireachIC3`. It should be noted that IPP erroneously reports UNSAT for most of the problems from the PARCPRINTER and WOODWORKING domains and we counted this as failures. Because `minireachIC3`, on the other hand, solves most of the problems from these domains, its score should be lowered by 94 problems to obtain a "fair" comparison on the problem set which excludes these two domains.

Notice that the performance of IPP with 466 solved problems is quite low compared to the best configuration of PDRplan1.1, which solves 1212 problems within 180 seconds. This raises the question whether Graphplan could be improved by enhancing it with the obligation rescheduling trick. We were able to confirm this experimentally. A relatively straightforward modification of IPP which retries a candidate goal set at time $t + 1$ after it has failed at time t was able to solve 676 problems.[34] Thus obligation rescheduling can be seen as an answer to the long standing question posed in the last remark of the original Graphplan paper by Blum and Furst (1997), i.e., as a way to trade plan quality for speed.

7. Discussion: a Closer Look at Two Domains

The fact that PDR maintains its reachability information organized in layers and uses the simple language of propositional clauses (CNF) to express the corresponding constraints often allows us to obtain additional insights on how the algorithm traverses the search space by inspecting the layers generated for concrete problems. This is especially rewarding in cases where PDR seems to be struggling with a relatively simple problem, as it often leads to a discovery of ideas for future improvements. In this section we have a closer look at the behavior of PDR on two simple domains. We conjecture that the algorithm could be improved by employing a more expressive constraint formalism than CNF.

7.1 1998-LOGISTICS

The task in the LOGISTICS domain is to transport packages between locations. Locations belong to cities and within a city trucks may be used to move packages with the help of the load-truck, drive-truck, and unload-truck actions. Additionally, some of the locations are designated as airports and airplanes may be used to transport packages between airports possibly across cities via the load-airplane, fly-airplane, and unload-airplane actions.

Although the LOGISTICS domain is generally considered to be a simple one, Table 3 (page 304) reveals relatively poor performance of PDRplan on LOGISTICS problems. Here are two of our initial findings from the inspection of the layer clauses generated by PDR, which shed some light on what is going on "under the hood".

34. Also the performance of the corresponding configuration of `minireachIC3` goes up from the mentioned 484 to 733 solved problems within 180 seconds when obligation rescheduling is turned on.

- PDR often generates very long clauses.

  Because there are typically many distinct (although similar) ways to achieve a subgoal and all of them need to be taken into account, large reason sets are computed and subsequently long clauses derived. For example, if a package needs to be transported from one city to another, any of the available airplanes can potentially be used for that purpose. We often encounter derived clauses like

  $$\mathit{subg} \vee \mathit{at}(\mathit{apn}_1, \mathit{apt}) \vee \mathit{at}(\mathit{apn}_2, \mathit{apt}) \vee \ldots \vee \mathit{at}(\mathit{apn}_n, \mathit{apt}) \tag{7}$$

  expressing that if the subgoal *subg* has not been derived yet, at least one of the available airplanes apn_i need to be present at the airport *apt*.

- PDR generates many similar clauses.

  Even if an action has more than one precondition false in the current state, at most one of these preconditions is reflected in the computed reason of an unsuccessful extension. Thus with many actions available for achieving a subgoal, sometimes many clauses are needed as PDR tries to find the right achieving action and satisfy all its preconditions.

  In addition to the above clause (7) we could see PDR subsequently derive the following clauses in the same layer:

  $$\begin{aligned} &\mathit{subg} \vee \mathit{in}(\mathit{obj}, \mathit{apn}_1) \vee \mathit{at}(\mathit{apn}_2, \mathit{apt}) \vee \ldots \vee \mathit{at}(\mathit{apn}_n, \mathit{apt}), \\ &\mathit{subg} \vee \mathit{at}(\mathit{apn}_1, \mathit{apt}) \vee \mathit{in}(\mathit{obj}, \mathit{apn}_2) \vee \ldots \vee \mathit{at}(\mathit{apn}_n, \mathit{apt}), \\ &\qquad\qquad\qquad\qquad\qquad \cdots \\ &\mathit{subg} \vee \mathit{at}(\mathit{apn}_1, \mathit{apt}) \vee \mathit{at}(\mathit{apn}_2, \mathit{apt}) \vee \ldots \vee \mathit{in}(\mathit{obj}, \mathit{apn}_n). \end{aligned} \tag{8}$$

  Note that although the pattern indicates n different clauses, there are in the worst case 2^n clauses potentially derivable with the "arbitrary" choice between $\mathit{at}(\mathit{apn}_i, \mathit{apt})$ and $\mathit{in}(\mathit{obj}, \mathit{apn}_i)$ for every i.

Although we have so far described PDR as an algorithm based on propositional logic, we believe it could be generalized to take advantage of first order constraints. Consider the clause (7) above. An equivalent first order version (aware of the type *airplane*) would read

$$\mathit{subg} \;\vee\; \exists \mathit{Apn} \in \mathit{airplane} \,.\, \mathit{at}(\mathit{Apn}, \mathit{apt}),$$

which is much more succinct.[35] Moreover, it could potentially be derived by just analyzing the action schemes *unload-airplane*$(\mathit{Obj}, \mathit{Apn}, \mathit{Loc})$, ..., etc., instead of iterating through the much larger set of instantiated actions. Working out the missing details is an interesting direction for future research. An inspiration could be found in the work of Ranise (2013), whose setting of security policy analysis is very close to automated planning.

Another independent direction for enhancing the expressive power of the used constraints could be the introduction of "conjunctive literals". Notice that the set of clauses (8) is, in fact, subsumed by a single generalized clause

$$\mathit{subg} \vee \bigvee_{i=1}^{n} \mathit{in}(\mathit{obj}, \mathit{apn}_i) \wedge \mathit{at}(\mathit{apn}_i, \mathit{apt}),$$

35. Symbols starting with an uppercase letter, like *Apn*, stand for first order variables.

where we allow conjunctions in place of single literals. In this envisioned generalization of PDR, such conjunctions would naturally come from the precondition lists of actions, and their use could help with solving, e.g., the LOGISTICS problems more efficiently. Of course, there are again details that would need to be worked out.

7.2 1998-GRIPPER

GRIPPER is a very simple domain which models a robot with two grippers trying to move balls from one room to another. This domain is fully solved by PDRplan in the default configuration. In fact, although the individual problems differ in size, PDRplan is able (thanks to obligation rescheduling) to solve all of them during iteration 3 of the main loop.[36] The reason for this seems to be the virtual independence of the individual goals, which can be considered one be one by PDR. We conjecture that the algorithm solves problems from the GRIPPER domain in polynomial time.

Despite the simplicity, GRIPPER is known to be difficult to solve optimally by heuristic search planners (see Helmert & Röger, 2008). This also holds for PDR, which exhibits exponential behavior when attempting to find a minimal length plan, i.e., when run with obligation rescheduling (and sidestepping) turned off. To demonstrate the reason, let us abstract and simplify GRIPPER a bit more and consider a domain in which the task is to achieve n independent goals from the set $\{g_1, \ldots, g_n\}$, such that achieving a particular goal is trivial, but the individual goals can only be achieved one by one.

On such a domain, PDR will eventually need to express via layer L_i that *at least* $(n-i)$ goals should already be achieved. Such a "counting" constraint has inherently large clausal description. Namely, the set L_i takes the form

$$\bigwedge g_{j_0} \vee \ldots \vee g_{j_i},$$

where the conjunction ranges over all $(i+1)$-element subsets $\{j_0, \ldots, j_i\}$ of $\{1, \ldots, n\}$. The size of the layer L_i is, therefore, proportional to the binomial coefficient $\binom{n}{i+1}$, which, in particular, means that the size of the layer $L_{\lfloor n/2 \rfloor}$ grows exponentially with n.

As already suggested by Helmert and Röger (2008) this phenomenon could be overcome by exploiting symmetries (Fox & Long, 1999) inherently present in the problem. This could be particularly rewarding in PDR, where the layer clauses (although derived as a response to unsuccessful extensions of arbitrary reachable states) logically depend only on the goal condition G, where the symmetries typically reside. Thus unlike Fox and Long (1999), who define symmetric objects to be those which are indistinguishable from one another in terms of their initial and goal configuration, one could with PDR use a stronger notion of symmetry derived from the goal condition only.

8. Conclusion

In this paper we have examined PDR, a novel algorithm for analyzing reachability in symbolic transition systems, from the perspective of automated planning. Our main contribution lies in recognizing that a part of the algorithm's work normally delegated to a

36. Other domains fully solved by PDRplan during a particular fixed iteration are 2002-ZENOTRAVEL (iteration 3), 2004-PHILOSOPHERS (iteration 6), and 2006,2008,2011-OPENSTACKS (iteration 4).

SAT-solver can, in the context of planning, be implemented directly by a polynomial time procedure. We have experimentally confirmed that this modification, as well as several other proposed improvements, boost the performance of PDR on planning benchmarks. Our implementation of the algorithm called PDRplan was able to compete respectably with state of the art planners, solving most problems on several domains.

Despite the already promising results, there is still room for further development. One direction is work on extending PDRplan towards richer planning formalisms. For example, we believe the `extend` procedure can be enhanced to cope with conditional effects of actions in a straightforward way. Efficiently dealing with action costs or domain axioms could turn out to be more difficult. Another promising direction is the idea to generalize PDR to a more expressive constraint language than CNF. While it is clear that stronger constraints imply better guidance towards the goal, devising an efficient method for combining new constraints from old ones is obviously a challenging task. It seems, however, that this "departure beyond the clausal level" could have a simpler solution inside the planning-specific framework of the `extend` procedure than it, perhaps, has within general purpose constraint solvers.

Acknowledgments

We thank Jussi Rintanen for useful comments and remarks, as well as for the help with the Mp encoder. We also thank Tomáš Balyo for providing us with the SASE encoding tool. Finally, we want to thank the anonymous reviewers for their insightful suggestions and Malte Helmert for the help with the preparation of the final version of this text.

This research was partially supported by the Czech Science Foundation under the project P103-10-1287.

Appendix A. Pseudocode of the Improvements of Section 4.5

Pseudocode 6 displays stage one of procedure $\texttt{extend}^+$, an enhancement of the `extend` procedure by the lazy false clause computation technique (Section 4.5.1) and with a support for sidestepping (Section 4.5.2). Stage two of $\texttt{extend}^+$ is meant to be supplemented from the same stage of the original `extend` procedure (Pseudocode 3) and stage three employes inductive minimization as described in Section 4.2 (Pseudocode 4).

Pseudocode 7 details the workings of PDRplan1.1. With the help of the $\texttt{extend}^+$ procedure it realizes sidestepping (Section 4.5.2). Moreover, it incorporates the technique for keeping obligations between iterations (Section 4.5.3). The clause propagation phase is identical to the one already presented in Pseudocode 5.

Pseudocode 6 Stage one of $\texttt{extend}^{+}(s, i)$:

Input:

Obligation (s, i), i.e. a state s and an index i, such that $s \not\models L_{i-1}$

Output:

Either an obligation $(t, i-1)$ where t is a successor of s and $t \models L_{i-1}$, or
an obligation (t, i) where t is a successor of s, $t \models L_i$ and
t satisfies strictly more clauses from L_{i-1} than s, or
an inductive reason $r \subseteq \mathit{Lits}(s)$

$L^s \leftarrow \{c \in L_{i-1} \mid s \not\models c\}$ /* Clauses false in s */
$R_{noop} \leftarrow \{\neg c \mid c \in L^s\}$ /* The reasons for the noop action */
assert $R_{noop} \neq \emptyset$ /* Follows from the contract with the caller */
$\mathcal{R} \leftarrow \{R_{noop}\}$ /* The set of reason sets collected so far */
$a_{side} \leftarrow \mathit{noop}$ /* Current best candidate for sidestepping (*noop* as a dummy value) */
$x_{side} \leftarrow |L^s|$ /* Score of the current best candidate */

foreach $a \in \mathcal{A}$ **do**
  $\mathit{pre}^s_a \leftarrow \{l \in \mathit{pre}_a \mid s \not\models l\}$ /* Preconditions false in s */
  $t \leftarrow \mathit{apply}(s, (\emptyset, \mathit{eff}_a))$ /* Ignore the preconditions and apply the effects of a */
  $L^{s,t} \leftarrow \{c \in L^s \mid t \not\models c\}$ /* The lazy approach: clauses false both in s and t */
  if $L^{s,t} = L^s$ **then** /* No improvement over s */
    continue /* Do nothing: the reason set would be subsumed by R_{noop} */

  if $\mathit{pre}^s_a = \emptyset$ **and** $|L^{s,t}| < x_{side}$ **then** /* The action is promising … */
    $L^t \leftarrow \{c \in L_{i-1} \mid t \not\models c\}$ /* … we must compute the full L^t */
    if $L^t = \emptyset$ **then**
      return $(t, i-1)$ /* The positive part: returning a true successor */
    if $L^t = L^{s,t}$ and $t \models L_i$ **then** /* No false clauses besides those from $L^{s,t}$ */
      $a_{side} \leftarrow a$
      $x_{side} \leftarrow |L^{s,t}|$
  else
    $L^t \leftarrow L^{s,t}$ /* Save time by using $L^{s,t}$ instead of the full L^t below */

  $L^t_0 \rightarrow \{c \in L^t \mid c \cap \mathit{pre}^s_a = \emptyset\}$ /* False clauses with a non-subsumed reason */
  $R_a \leftarrow \{\{\neg l\} \mid l \in \mathit{pre}^s_a\} \cup \{\{\neg l \mid l \in c \text{ and } \neg l \notin \mathit{eff}_a\} \mid c \in L^t_0\}$
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{R_a\}$ /* Record the reason set */

if $x_{side} < |L^s|$ **then**
  assert $a_{side} \neq \mathit{noop}$
  return $(\mathit{apply}(s, a_{side}), i)$ /* Successfully sidestepping with the best candidate */

/* Continue with stage two as in Pseudocode 3 and stage three as in Pseudocode 4 */

Pseudocode 7 Algorithm `PDRplan1.1`$(X, s_I, G, \mathcal{A})$:

Input:

A positive STRIPS planning problem $\mathcal{P} = (X, s_I, g, \mathcal{A})$

Output:

A plan for $\mathcal{P}$ or a guarantee that no plan exists

```
L_0 ← {{p} | p ∈ g} /* The goal cube treated as a set of unit clauses */
foreach j > 0 : L_j ← ∅
Q ← {(s_I, 0)}
for k = 0, 1, ... do
  /* Path construction: */
  while there is (s, i) in Q with i ≤ k do
    pop some (s, i) from Q with minimal i
    if s ⊭ L_i then
      Q ← Q ∪ (s, i + 1)
    else if i = 0 then
      return PLAN FOUND
    else if extend⁺(s, i) returns an obligation (t, j) then
      assert j = i − 1 or j = i /* Either a regular extension or a sidestep */
      Q ← Q ∪ {(s, i), (t, j)}
    else
      extend⁺ returned a reason r ⊆ Lits(s)
      foreach 0 ≤ j ≤ i : L_j ← L_j ∪ {¬r}

      /* Obligation rescheduling: */
      Q ← Q ∪ {(s, i + 1)} /* Keep obligations with i + 1 > k till the next iteration */

  /* Clause propagation: */
  for i = 1, ..., k + 1 do
    foreach c ∈ L_{i−1} \ L_i do
      /* Clause push check */
      s_c ← {p ↦ 0 | p ∈ c} ∪ {p ↦ 1 | p ∈ (X \ c)}
      if for every a ∈ A : s_c ⊭ pre_a or apply(s_c, a) ⊭ L_{i−1} then
        L_i ← L_i ∪ {c}
    /* Convergence check */
    if L_{i−1} = L_i then
      return NO PLAN POSSIBLE
```

References

Bäckström, C., Jonsson, P., & Ståhlberg, S. (2013). Fast detection of unsolvable planning instances using local consistency. In Helmert, M., & Röger, G. (Eds.), *SOCS*. AAAI Press.

Bäckström, C., & Nebel, B. (1995). Complexity results for SAS^+ planning. *Computational Intelligence*, *11*, 625–656.

Balyo, T., Bardiovský, V., Dvořák, F., & Toropila, D. (2012). Freelunch planning library. Available at `http://ktiml.mff.cuni.cz/freelunch/`.

Balyo, T., & Chrpa, L. (2014). Eliminating all redundant actions from plans using SAT and MaxSAT. In *ICAPS 2014 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*. To appear.

Biere, A., Heljanko, K., Seidl, M., & Wieringa, S. (2012). Hardware model checking competition 2012. Web site, `http://fmv.jku.at/hwmcc12/`.

Blum, A., & Furst, M. L. (1997). Fast planning through planning graph analysis. *Artif. Intell.*, *90*(1–2), 281–300.

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artif. Intell.*, *129*(1–2), 5–33.

Bradley, A. R. (2011). SAT-based model checking without unrolling. In Jhala, R., & Schmidt, D. A. (Eds.), *VMCAI*, Vol. 6538 of *Lecture Notes in Computer Science*, pp. 70–87. Springer.

Bradley, A. R., & Manna, Z. (2007). Checking safety by inductive generalization of counterexamples to induction. In *FMCAD*, pp. 173–180. IEEE Computer Society.

Clifford, R., & Popa, A. (2011). Maximum subset intersection. *Inf. Process. Lett.*, *111*(7), 323–325.

Domshlak, C., Helmert, M., Karpas, E., Keyder, E., Richter, S., Röger, G., Seipp, J., & Westphal, M. (2011). BJOLP: The big joint optimal landmarks planner. In *Seventh International Planning Competition (IPC 2011), Deterministic Part*, pp. 91–95.

Eén, N., Mishchenko, A., & Brayton, R. K. (2011). Efficient implementation of property directed reachability. In Bjesse, P., & Slobodová, A. (Eds.), *FMCAD*, pp. 125–134. FMCAD Inc.

Eén, N., & Sörensson, N. (2003). An extensible SAT-solver. In Giunchiglia, E., & Tacchella, A. (Eds.), *SAT*, Vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518. Springer.

Fox, M., & Long, D. (1999). The detection and exploitation of symmetry in planning problems. In Dean, T. (Ed.), *IJCAI*, pp. 956–961. Morgan Kaufmann.

Gazen, B. C., & Knoblock, C. A. (1997). Combining the expressivity of UCPOP with the efficiency of Graphplan. In Steel, S., & Alami, R. (Eds.), *ECP*, Vol. 1348 of *Lecture Notes in Computer Science*, pp. 221–233. Springer.

Ghallab, M., Nau, D. S., & Traverso, P. (2004). *Automated planning – theory and practice*. Elsevier.

Haas, A. R. (1987). The case for domain-specific frame axioms. In *The Frame Problem in Artificial Intelligence, Proceedings of the 1987 Workshop on Reasoning about Action*. Morgan Kaufmann.

Helmert, M. (2006). The Fast Downward planning system. *J. Artif. Intell. Res. (JAIR)*, *26*, 191–246.

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

Helmert, M., Do, M., & Refanidis, I. (2008). IPC 2008, deterministic part. Web site, `http://ipc.informatik.uni-freiburg.de`.

Helmert, M., Haslum, P., & Hoffmann, J. (2007). Flexible abstraction heuristics for optimal sequential planning. In Boddy, M. S., Fox, M., & Thiébaux, S. (Eds.), *ICAPS*, pp. 176–183. AAAI.

Helmert, M., & Röger, G. (2008). How good is almost perfect?. In Fox, D., & Gomes, C. P. (Eds.), *AAAI*, pp. 944–949. AAAI Press.

Hoffmann, J., Kissmann, P., & Álvaro Torralba (2014). "Distance"? Who cares? Tailoring Merge-and-Shrink heuristics to detect unsolvability. In *ICAPS 2014 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*. To appear.

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res. (JAIR)*, *14*, 253–302.

Howey, R., Long, D., & Fox, M. (2004). VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *ICTAI*, pp. 294–301. IEEE Computer Society. Software available at `http://www.plg.inf.uc3m.es/ipc2011-deterministic/Resources`.

Huang, R., Chen, Y., & Zhang, W. (2012). SAS$^+$ planning as satisfiability. *J. Artif. Intell. Res. (JAIR)*, *43*, 293–328.

IPC (2014). International planning competition. Web site, `http://ipc.icaps-conference.org/`, accessed 19/05/2014.

Johnson, W. W., & Story, W. E. (1879). Notes on the "15" puzzle. *American Journal of Mathematics*, *2*(4), 397–404.

Kautz, H., Selman, B., & Hoffmann, J. (2006). SatPlan: Planning as satisfiability. In *Working Notes of the 5th International Planning Competition, Cumbria, UK*. Software available at `http://www.cs.rochester.edu/~kautz/satplan/`.

Kautz, H. A., McAllester, D. A., & Selman, B. (1996). Encoding plans in propositional logic. In Aiello, L. C., Doyle, J., & Shapiro, S. C. (Eds.), *KR*, pp. 374–384. Morgan Kaufmann.

Kautz, H. A., & Selman, B. (1992). Planning as satisfiability. In *ECAI*, pp. 359–363.

Kautz, H. A., & Selman, B. (1996). Pushing the envelope: Planning, propositional logic and stochastic search. In Clancey, W. J., & Weld, D. S. (Eds.), *AAAI/IAAI, Vol. 2*, pp. 1194–1201. AAAI Press / The MIT Press.

Koehler, J. (1999). *IPP – A Planning System for ADL and Resource-Constrained Planning Problems*. Habiliation thesis, University of Freiburg.

Long, D., & Fox, M. (1999). Efficient implementation of the plan graph in STAN. *J. Artif. Intell. Res. (JAIR)*, *10*, 87–115.

Massey, B. (1999). *Directions In Planning: Understanding The Flow Of Time In Planning*. Ph.D. thesis, University of Oregon.

McCarthy, J., & Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., & Michie, D. (Eds.), *Machine Intelligence 4*, pp. 463–502. Edinburgh University Press.

Nakhost, H., & Müller, M. (2010). Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In Brafman, R. I., Geffner, H., Hoffmann, J., & Kautz, H. A. (Eds.), *ICAPS*, pp. 121–128. AAAI.

Pettersson, M. P. (2005). Reversed planning graphs for relevance heuristics in AI planning. In *Planning, Scheduling and Constraint Satisfaction: From Theory to Practice*, Vol. 117 of *Frontiers in Artificial Intelligence and Applications*, pp. 29–38. IOS Press.

Ranise, S. (2013). Symbolic backward reachability with effectively propositional logic – applications to security policy analysis. *Formal Methods in System Design*, *42*(1), 24–45.

Richter, S., & Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res. (JAIR)*, *39*, 127–177.

Rintanen, J. (1998). A planning algorithm not based on directional search. In Cohn, A. G., Schubert, L. K., & Shapiro, S. C. (Eds.), *KR*, pp. 617–625. Morgan Kaufmann.

Rintanen, J. (2004). Evaluation strategies for planning as satisfiability. In de Mántaras, R. L., & Saitta, L. (Eds.), *ECAI*, pp. 682–687. IOS Press.

Rintanen, J. (2008a). Planning graphs and propositional clause-learning. In Brewka, G., & Lang, J. (Eds.), *KR*, pp. 535–543. AAAI Press.

Rintanen, J. (2008b). Regression for classical and nondeterministic planning. In Ghallab, M., Spyropoulos, C. D., Fakotakis, N., & Avouris, N. M. (Eds.), *ECAI*, Vol. 178 of *Frontiers in Artificial Intelligence and Applications*, pp. 568–572. IOS Press.

Rintanen, J. (2012). Planning as satisfiability: Heuristics. *Artif. Intell.*, *193*, 45–86.

Rintanen, J., Heljanko, K., & Niemelä, I. (2006). Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, *170*(12–13), 1031–1080.

Suda, M. (2013a). DIMSPEC, a format for specifying symbolic transition systems. Web site, `http://www.mpi-inf.mpg.de/~suda/DIMSPEC.html`.

Suda, M. (2013b). Duality in STRIPS planning. *CoRR*, *abs/1304.0897*.

Suda, M. (2014). Property directed reachability for automated planning. Web site, `http://www.mpi-inf.mpg.de/~suda/PDRplan.html`.

Wikipedia (2014). 15 puzzle — wikipedia, the free encyclopedia. Web site, `http://en.wikipedia.org/wiki/15_puzzle`, accessed 19/05/2014.