# Automated Planning using Property-Directed Reachability with Seed Heuristics

Master Thesis

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

Examiner: Prof. Dr. Malte Helmert
Supervisors: Dr. Salomé Eriksson and Simon Dold

Tim Bachmann
tim.bachmann@stud.unibas.ch
15-916-299

May 06, 2023

## Abstract

Planning is the process of finding a path in a planning task from the initial state to a goal state. Multiple algorithms have been implemented to solve such planning tasks, one of them being the Property-Directed Reachability algorithm. Property-Directed Reachability utilizes a series of propositional formulas called layers to represent a super-set of states with a goal distance of at most the layer index. The algorithm iteratively improves the layers such that they represent a minimum number of states. This happens by strengthening the layer formulas and therefore excluding states with a goal distance higher than the layer index. The goal of this thesis is to implement a pre-processing step to seed the layers with a formula that already excludes as many states as possible, to potentially improve the run-time performance. We use the pattern database heuristic and its associated pattern generators to make use of the planning task structure for the seeding algorithm. We found that seeding does not consistently improve the performance of the Property-Directed Reachability algorithm. Although we observed a significant reduction in planning time for some tasks, it significantly increased for others.

# Contents

**1 Introduction** **3**

**2 Background** **4**
2.1 Propositional Logic . . . 4
2.2 Symbolic Transition Systems . . . 5
2.3 Planning . . . 5
2.3.1 Finite-Domain Representation . . . 6
2.4 Heuristics . . . 7
2.4.1 Pattern Database Heuristic . . . 7

**3 Property-Directed Reachability** **9**
3.1 Data Structures . . . 9
3.2 The Algorithm . . . 11
3.3 Extending the Witnessing Path . . . 16

**4 Heuristics in PDR** **19**
4.1 Seeding Layers . . . 19
4.2 Layers from Heuristics . . . 20
4.3 Layers from Pattern Database Heuristics . . . 20

**5 Implementation** **24**
5.1 PDR in the Fast Downward Planning System . . . 24
5.1.1 Data Structures . . . 24
5.2 Heuristic Seeding of Layers . . . 26
5.2.1 The Pattern Database PDR Heuristic . . . 26
5.2.2 Seeding a Layer from the Pattern Database . . . 27

**6 Evaluation** **28**
6.1 Evaluated Configurations . . . 28
6.2 Benchmark Setup . . . 29
6.3 Results . . . 30

**7 Conclusion** **36**

**A Appendix** **39**

# 1 Introduction

Many problems in everyday life can be modeled as planning problems that can be solved by dedicated software. While everyday problems are usually non-deterministic, modeling those problems as a simplified deterministic planning problem is often possible. A planning problem consists of a state space with a single initial state and a set of goal states. The act of planning refers to the process of finding a path from the initial state to one of the goal states. Such a path encoded as a sequence of operators is called a plan.

The Property-Directed Reachability algorithm can be used to solve such planning problems. One of the main components of the algorithm is a sequence of formulas $L_0, L_1, \ldots$, referred to as layers, where each layer $L_i$ represents a super-set of states that are at most $i$ steps away from a goal state. Property-Directed Reachability iteratively modifies each layer to restrict the number of states represented by it, in a way such that the set of states represented by the layer $L_i$ approaches the set of states that is at most $i$ steps away from a goal state. The algorithm initializes the layers $L_i$ with $i > 0$ with the formula $\top$, which represents the whole state space.

The algorithm was first introduced under the name IC3 ("Incremental Construction of Inductive Clauses for Indubitable Correctness") by Bradley as a form of SAT-based model checking [1]. The name Property-Directed Reachability was introduced by Eén et al. as part of their work on a simplified and faster implementation [4]. Suda adapted the algorithm for planning and proposed a SAT-free version for positive STRIPS planning tasks [13].

Heuristics are functions that efficiently estimate the goal cost of a state. Often this corresponds to the length of a path from this state to one of the goal states. Some heuristics, like the pattern database heuristic, achieve this by deriving a simplified planning task and solving it in a pre-computation step. The pattern database heuristic in particular has some favorable properties, such as being admissible, meaning it never overestimates the cost, and exposing the simplified planning task as a set of variables.

The goal of this thesis is to explore ways to initialize, or seed, the Property-Directed Reachability algorithm with more restrictive layers, with the intuition that those pre-seeded layers improve the performance of the algorithm significantly. We propose to extract such formulas from the pattern database heuristic. In addition, we explore the run-time effects of seeding the layers.

This thesis is organized into seven chapters. Chapter 2 is an introduction to the necessary concepts and terminology used throughout the thesis. In Chapter 3, we describe the Property-Directed Reachability algorithm, as proposed by Suda, in detail. In Chapter 4 we propose a general seeding method using a generic heuristic, and a specific method based on the pattern database heuristic. In Chapter 5, we describe how we implemented the algorithm and the layer seeding in the Fast Downward planning system. The performance of the seeded and non-seeded variants of the algorithm are compared in Chapter 6. Finally, in Chapter 7, we summarize our findings and provide suggestions for future research.

# 2 Background

In this chapter we will explain the concepts and terminology used in this thesis. A brief introduction to propositional logic, symbolic transition systems, the concepts of planning, and heuristics, specifically the pattern database heuristic, follows. Additionally, a thorough explanation of the Property-Directed Reachability algorithm follows in Chapter 3.

## 2.1 Propositional Logic

In this section we will define some concepts of propositional logic, used in the rest of the thesis.

We call a set of variables a signature $\Sigma$. A literal is either a variable $v \in \Sigma$ or its negation $\neg v$. The function $var(l) = v$ where $l = v$ or $l = \neg v$, returns the variable that is used to build the literal $l$.

A formula is an expression consisting of either a literal $l$, the truth constants $\top$, which always evaluates to **true**, or $\bot$ which always evaluates to **false**, or one of the following connectives: A conjunction $\phi \wedge \psi$, with $\phi$ and $\psi$ being formulas. A disjunction $\phi \vee \psi$, with $\phi$ and $\psi$ being formulas. Or a negation $\neg\phi$, with $\phi$ being a formula.

A clause is a formula consisting of disjunctions of literals, and a cube is a formula consisting of conjunctions of literals. A literal is called positive if it is a variable that is not negated, and negative if the negation operator is applied. When multiple negation operators are applied consecutively, we silently remove them pairwise. Clauses and cubes are called positive if they only contain positive literals, and negative if they only contain negative literals. Clauses and cubes can also be represented as sets of literals, where the formula is equal to the disjunction or the conjunction of the elements respectively. A set of clauses denotes the conjunction of those clauses and thus represents a formula in conjunctive normal form (CNF), and in turn, a set of cubes refers to a formula in disjunctive normal form (DNF). An assignment is a function $s : \Sigma \to \{\top, \bot\}$ that maps variables to truth values. A partial assignment maps a subset of all variables to a truth value, while a full assignment applies to all variables in $\Sigma$. We write an assignment in the form $s = \{(v_1 \to d_1), \ldots, (v_n \to d_n)\}$, where $v_1, \ldots, v_n$ are variables and $d_1, \ldots, d_n$ are the assigned values.

We write $s \models \phi$ if an assignment $s$ satisfies the formula $\phi$. The satisfy relation between an assignment $s$ and a formula $\phi$ is defined recursively as follows:

$$\begin{aligned}
&\textbf{if } \phi = v \text{ where } v \text{ is a variable}, \textbf{ then } s \models \phi \text{ iff } s(v) = \top \\
&\textbf{if } \phi = \neg\psi_1, \textbf{ then } s \models \phi \text{ iff not } s \models \psi_1 \\
&\textbf{if } \phi = \psi_1 \wedge \psi_2, \textbf{ then } s \models \phi \text{ iff } s \models \psi_1 \text{ and } s \models \psi_2 \\
&\textbf{if } \phi = \psi_1 \vee \psi_2, \textbf{ then } s \models \phi \text{ iff } s \models \psi_1 \text{ or } s \models \psi_2
\end{aligned}$$

A formula is called SAT or *satisfiable* if there exists an assignment that satisfies the formula, and UNSAT or *unsatisfiable* if no such assignment exists. We define the implication symbol "$\Rightarrow$" as a relation between two formulas. The implication $G \Rightarrow F$ between the formulas $G$ and $F$ means that every assignment that satisfies $G$ also satisfies $F$. This is equivalent to the statement "$G \wedge \neg F$ is UNSAT".

The function *Lits* over a partial assignment $s$, where $\Sigma' \subseteq \Sigma$ contains the assigned variables, is defined as the cube

$$Lits(s) = \{v \mid v \in \Sigma' \wedge s(v) = \top\} \cup \{\neg v \mid v \in \Sigma' \wedge s(v) = \bot\},$$

which maps each variable $v_1$ assigned to $\top$ to the positive literal $v_1$ and each variable $v_2$ assigned to $\bot$ to the negative literal $\neg v_2$. All assignments that agree with $s$ satisfy the formula $Lits(s)$.

## 2.2 Symbolic Transition Systems

A symbolic transition system (STS) is defined as the tuple $\mathcal{S} = (\Sigma, I, G, T)$ with the following elements: The signature $\Sigma = \{a, b, \dots\}$ is a finite set of propositional variables at the current point in time. An assignment $s : \Sigma \rightarrow \{\top, \bot\}$ is a mapping from variables to truth values. A full assignment is called a state. We use a disjunct copy of $\Sigma$ denoted by $\Sigma' = \{a', b', \dots\}$ to describe the state of the system after one step. $I$ is the initial formula, a predicate logic formula over $\Sigma$, representing the set of initial states. In this thesis, we will only consider transition systems with a single initial state $s_I \models I$. $G$ is the goal formula over $\Sigma$, which represents the set of goal states. $T$ is the transition formula over $\Sigma \cup \Sigma'$. It is a CNF formula where the variables in $\Sigma$ represent the current state and the variables in $\Sigma'$ represent the next state. A transition from an assignment $s$ to an assignment $s'$ is valid for a given STS, if $s \models T$ with regards to $\Sigma$ and $s' \models T$ with regards to $\Sigma'$.

The STS is an alternative but equivalent representation of an explicit transition system $\mathcal{T} = (S, s_I, s_G, t)$ where $S$ is the finite set of states where each state is an assignment of all variables in $\Sigma$. The finite set of initial states $s_I \subseteq S$ consists of all states that are a model of $I$. The finite set of goal states $s_G \subseteq S$ consists of all states that are a model of $G$. The set of transition relations $t \subseteq S \times S'$ contains pairs of states that jointly satisfy the transition formula.

## 2.3 Planning

A unit cost STRIPS planning task is defined as a tuple $\Pi = (\mathcal{V}, \mathcal{O}, s_I, s_*)$ [5], which implicitly defines a symbolic transition system. $\mathcal{V}$ is a finite set of variables $V \in \mathcal{V}$, corresponding directly to the elements of the signature $\Sigma$ in the STS. A state $s_a$ in a STRIPS planning task is a full assignment $a : \mathcal{V} \rightarrow \{\top, \bot\}$. We call the set of all possible states $S$. The finite set $\mathcal{O}$ contains operators $o \in \mathcal{O}$ with a precondition $pre(o)$ and an effect $eff(o)$. Both the precondition and the effect can be modeled as a cube over a subset of $\mathcal{V}$. An operator is applicable on a state $s$ if $s \models pre(o)$. Applying an applicable operator $o$ on a state $s$ results in the successor state $t = s[\![o]\!]$, which assigns all positive variables in $eff(o)$ to $\top$, all negative variables in $eff(o)$ to $\bot$, and all remaining variables to the assignment of $s$. Aggregating the precondition and the effect as a formula in CNF results in a formula that is equivalent to the transition formula $T$ in the STS

$$T \equiv \bigvee_{o \in \mathcal{O}} \left(pre(o) \wedge (eff(o))'\right).$$

The state $s_I$ is the initial state, which is the only state that models the initial formula $I$ from the STS[1]. The goal description $s_*$ is a partial variable assignment,

[1]As we defined the initial formula $I$ to only be model of a single state.

corresponding to the goal formula in the STS. Operators can be chained together to form a sequence $\pi = \langle o_0, \dots o_n \rangle$. Such a sequence of operators $\pi$ is applicable on a state $s_0$ if every operator $o_i$ is applicable on the state $s_i$ resulting in the state $s_{i+1}$. If $\pi$ is applicable to the state $s_I$, and the resulting state $s_n$ satisfies $Lits(s_*)$, then $\pi$ is called a *plan* and the sequence of states $\sigma = \langle s_I, s_1, \dots, s_n \rangle$ is called a *witnessing path*. Most definitions of STRIPS tasks include a cost function on operators. We are however only considering tasks where each operator has a fixed cost of one and thus omit the cost function. Therefore, the cost of a plan is always equal to its length.

**Positive STRIPS Tasks** A STRIPS task is called *positive* if it fulfills the following set of requirements: The precondition $pre(o)$ of every operator $o \in \mathcal{O}$ is a positive cube. The goal $s_*$ is a partial assignment that only assigns to the value $\top$.

Most literature uses the definition of the positive STRIPS task synonymously with STRIPS. For the rest of this thesis, we will only consider positive STRIPS tasks.

### 2.3.1 Finite-Domain Representation

A planning task in Finite-Domain Representation (FDR) as introduced by Helmert [8] is defined by a tuple

$$\hat{\Pi} = \left(\hat{\mathcal{V}}, \hat{\mathcal{O}}, \hat{s}_I, \hat{s}_*\right),$$

with $\hat{\mathcal{V}}$ being a finite set of variables. While a variable of a STRIPS task can only be assigned to the values $\top$ and $\bot$, every variable $\hat{v} \in \hat{\mathcal{V}}$ has an associated finite domain $D_{\hat{v}}$ and can be assigned to any element of this domain. A partial assignment to of a single variable in the form $(\hat{v} \to d)$ is called a fact. A state in an FDR task is a full assignment $\hat{s}$. Similar to a STRIPS task, $s_I$ refers to the initial state and $s_*$ is the partial assignment that describes the set of goal states. The finite set $\hat{\mathcal{O}}$ contains operators $\hat{o} \in \hat{\mathcal{O}}$ with a precondition $pre(\hat{o})$ and an effect $eff(\hat{o})$. The precondition is a cube over facts, while the effect is a partial assignment. Similar to STRIPS, an operator $\hat{o}$ is applicable to a state $s$ when $s \models pre(\hat{o})$. The resulting successor state $t = s[\![\hat{o}]\!]$ assigns all variables $\hat{v}$ to a value $d_{\hat{v}}$ if $(\hat{v} \to d_{\hat{v}}) \in eff(\hat{o})$. For all variables that are not assigned by the effect, $t$ assigns them to the same value as $s$. For an FDR variable $\hat{v} \in \hat{\mathcal{V}}$ the function $var$, as defined in Section 2.1, returns the fact on which a literal is based.

**Converting an FDR Task into a STRIPS Task** An FDR planning-task $\hat{\Pi} = \left(\hat{\mathcal{V}}, \hat{s}_I, \hat{s}_*, \hat{\mathcal{O}}\right)$ can be converted to a STRIPS planning task $\Pi = (\mathcal{V}, \mathcal{O}, s_I, s_*)$. The set of STRIPS variables is defined as the set of tuples containing the FDR variable and its assigned value $\mathcal{V} = \{(\hat{v}, d) \mid \hat{v} \in \hat{\mathcal{V}}, d \in D_v\}$. The initial state is defined as the full assignment

$$s_I = \{(\hat{v}, d) \to \top \mid (\hat{v} \to d) \in \hat{s}_I\} \cup \{(\hat{v}, d) \to \bot \mid (\hat{v} \to d) \notin \hat{s}_I\}.$$

Similarly, the set of goal states is defined as the partial assignment

$$s_* = \{(\hat{v}, d) \to \top \mid (\hat{v} \to d) \in \hat{s}_*\}.$$

Every FDR operator $\hat{o} \in \hat{\mathcal{O}}$ has a corresponding STRIPS operator $o \in \mathcal{O}$ with a precondition consisting of the cube

$$pre(o) = \{(\hat{v}, d) \mid (\hat{v} \to d) \in pre(\hat{o})\}$$

and an effect consisting of the cube

$$\mathit{eff}(o) = \{(\hat{v}, d) \mid (\hat{v} \to d) \in \mathit{eff}(\hat{o})\} \cup \{\neg(\hat{v}, d) \mid (\hat{v} \to x) \in \mathit{eff}(\hat{o}), d \in D_{\hat{v}}, x \neq d\}.$$

As an example consider the following effect of an FDR task: $\mathit{eff}(\hat{o}) = \{(v \to d_1)\}$, where $D_v = \{a, b, c\}$. The effect in the converted STRIPS task is defined as $\mathit{eff}(o) = \{(v, a), \neg(v, b), \neg(v, c)\}$.

While the number of states increases during the conversion, the resulting task has equivalent plans to the original task. However, the conversion to STRIPS is not free of information loss. Due to the increase in the number of states during the conversion, the STRIPS task contains states that are not representable in the original FDR task. An FDR task with a variable $v$ with the domain $D_v = \{a, b\}$ results in a STRIPS task with the variables $(v, a)$, and $(v, b)$. While states containing the assignments $\{((v, a) \to \top), ((v, b) \to \bot)\}$ and $\{((v, a) \to \bot), ((v, b) \to \top)\}$ are representable in the FDR task as the assignments $\{(v \to a)\}$ and $\{(v \to b)\}$, all states containing the partial assignments $\{((v, a) \to \top), ((v, b) \to \top)\}$ and $\{((v, a) \to \bot), ((v, b) \to \bot)\}$ are not. However, those states that are unrepresentable in the original FDR task are still unreachable from the initial state in the STRIPS task. This means, that there is no sequence of operators $\langle o_1, o_2, \ldots, o_n \rangle$ that, when applied sequentially from the initial state $s_I[\![o_1]\!][\![o_2]\!] \ldots [\![o_n]\!]$, results in such a state. A group of such variables, where only one of them can ever be assigned to $\top$ in any reachable STRIPS state, is called mutual exclusive or *mutex*. The implicit information of what variables are mutex is lost in the conversion from an FDR task to a STRIPS task.

## 2.4 Heuristics

In the context of a STRIPS planning task, a heuristic function $h : S \to \mathbb{R} \cup \{\infty\}$ is an approximation of the goal distance of a state. For planing, there are several properties of heuristic functions. For this thesis we are mostly interested in heuristics that are admissible, meaning that the function never overestimates the distance to the goal.

For convenience, we define the heuristic function over a set of states $S' \subseteq S$ as the minimal heuristic value of any state in the set:

$$h(S') = \min_{s \in S'} \{h(s)\}$$

### 2.4.1 Pattern Database Heuristic

A pattern database heuristic (PDB) is a heuristic function $h_{pdb}$ that relies on a pre-computation step to calculate the heuristic values of states in a planning task [2]. This is accomplished by projecting the original state space onto a sufficiently smaller state space using a projection $\pi_P$ and calculating the cost for each projected state $s' \in \{\pi_P(s) \mid s \in S\}$, where $S$ refers to the set of all states. A projection $\pi_P : S \to S_P$ is a surjective function that uses a pattern $P$ to map a state $s$ to a corresponding state $s'$ in the smaller state space $S_P$.

The pattern $P \subseteq \mathcal{V}$ defines the variables that are considered in the projected state space. A projection $\pi_P$ maps two states $s_1$ and $s_2$ to the same projected state $s'$ if and only if all variables in $P$ are assigned to the same values in both states. If a pattern contains all variables $\mathcal{V}$, the projected state is the same as the non-projected state $s = \pi_{\mathcal{V}}(s)$.

The projected transition relation $t_\pi$ corresponds to the original transition relation $t$ where the "source" and "destination" states have been projected: $t_\pi = \{(\pi(s_s), \pi(s_d)) \mid (s_s, s_d) \in t\}$.

The precondition and the effect of an operator $o' \in \mathcal{O}'$ in the projected state space only considers the variables in the pattern $P$. The precondition and the effect are defined as

$$pre(o') = \{l \mid l \in pre(o) \text{ and } var(l) \in P\},$$

and

$$\mathit{eff}(o') = \{l \mid l \in \mathit{eff}(o) \text{ and } var(l) \in P\}.$$

The pre-computation step of the pattern database works by finding a suitable pattern $P$ for the projection $\pi_P$, such that all states $S_P$ of the projected state space fit into a predefined amount of memory while keeping as much of the complexity of the state space as possible. Next, a uniform-cost search starting from the goal states through the whole projected state space is performed. Every visited projected state gets recorded into the database along with the distance to the closest goal state. After the pre-computation step, the pattern database is initialized and can be used as any other heuristic function. Retrieving a heuristic value $h_{pdb}(s)$ of a state works by applying the same projection $\pi_P(s)$ as in the pre-computation to the state. The cost of the projected state gets queried from the database and is returned as the heuristic value.

One of the properties of the PDB is that it is admissible, which is a direct consequence of finding the optimal costs for each state in the smaller projected state space. PDBs are often defined on a planning task in FDR form since the finite-domain representation is more concise and allows for larger sub-tasks to be solved and stored in memory [8].

# 3 Property-Directed Reachability

Property Directed Reachability (PDR) is an algorithm that can be used for finding a witnessing path in symbolic transition systems [1][4]. It was originally designed for checking the correctness of hardware models. Suda showed that this algorithm can be used for planning [13]. This chapter will explain the PDR algorithm as presented by Suda. PDR performs an explicit search in a symbolic transition system which is complemented by symbolic reachability analysis using a SAT planner. In the context of planning, PDR can be used to compute an optimal plan in a given unit cost planning task.

The algorithm works by iteratively calculating a set of layers $L_i$. A layer $L_i$ is a CNF formula that represents a super-set of all states from which a goal state can be reached in $i$ or fewer steps. By repeatedly updating a layer and strengthening the formula, the algorithm progressively excludes more states from the layer such that as few states as possible are a model of the layer while still being the super-set of all states that are $i$ or fewer steps away from a goal. It then tries to find a witnessing path for the planning task by starting at the initial state and extending the path by only considering states that are modeling the corresponding layer.

PDR starts by initializing the layers. The layer $L_0$ is set to the goal formula and therefore is modeled by only the goal states. All other layers are initialized with an empty CNF formula. This formula is a tautology, meaning that all states in the transition system are a model of the layers $L_i$ with $i > 0$.

The algorithm then proceeds with the iteration which is counted by the variable $k$. Every iteration step goes through a path construction phase and a clause propagation phase. The path construction phase checks if $s_I$ is in the layer $L_k$ and when this is not the case skips to the clause propagation phase. Otherwise, it then tries to build a witnessing path $\langle s_I, s_1, \ldots, s_n, s_g \rangle$ to a goal state $s_g$, such that $s_I$ satisfies the layer $L_k$, every state $s_j$ satisfies $L_{k-j}$ for all $j \in \{1, \ldots, n\}$ and the state $s_g$ satisfies the layer $L_0$. Clause propagation then tries to strengthen the layers by propagating clauses from a layer $L_j$ to $L_{j+1}$. Both of those phases are described in detail below.

Suda proposes two variants of the PDR algorithm for planning: the original variant, and the SAT-solver-free variant. By removing the need for the SAT solver, Suda showed a performance increase, with the downside that the SAT-solver free algorithm is only defined for positive STRIPS tasks. Since the SAT-solver-free variant is specifically designed for planning, and the implementation introduced in Chapter 5 is based on that variant, the following explanation will focus on the SAT-solver-free variant.

## 3.1 Data Structures

**Layers** We represent each layer as a set of clauses, where the conjunction over the whole set builds the CNF. A state $s$ is considered to be "in" a layer $L$ if the state is a model of this layer $s \models L$. Figure 1 shows an example of a state space with three non-empty layers.

The PDR algorithm iteratively strengthens the layers by adding clauses to a layer and therefore restricting the set of states that are represented by it. However, as shown by Suda, the following invariants are always satisfied:

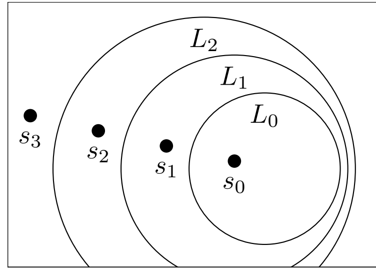



Figure 1: Layers $L_0, L_1, L_2$ and states $s_0, \dots, s_3$. The goal distance for each state $s_i$ is at least $i$. Every state $s_i$ is "in" all layers $L_j$ where $j \geq i$. The layers are depicted as sets of states that are represented by each layer.

1. $L_0$ represents exactly the goal states.
2. $L_{j+1} \subseteq L_j$; The layers are interpreted as sets of clauses. All clauses in a layer $L_{j+1}$ are also in the layer $L_j$, and therefore all states that are in layer $L_j$ are also in layer $L_{j+1}$.
3. $(L_j)' \wedge T \Rightarrow L_{j+1}$ for any $j \geq 0$; meaning that $L_{j+1}$ represents a super-set of states that have successors that are represented by $L_j$.

From the invariants follows that any state that is in a layer $L_i$ is also in every layer $L_j$ where $j > i$.

**Obligation** Obligations are used during the path construction phase and consist of a state $s$, an index $i \in \mathbb{N}$ as well as a reference to a parent obligation $q'$. During the iteration step $k$, an obligation $q = (s, q', i)$ is used in the path construction phase to store the fact that a path from the initial state to $s$ with length $k - i$ exists. This path is stored as the chain of parent obligations. The obligation of the initial state has the index $i = k$ and no parent obligation, we represent this empty reference by the symbol $*$. The set of obligations $Q$ acts as a priority queue, allowing access to the obligation which has the minimal index value $\min_{(s,q',i) \in Q} i$. This obligation contains the state which is the tip of the longest path from the initial state, where each state $s_j$ of the obligation $(s_j, q, j)$ is a model of the layer $L_j$.

Once the goal has been reached, meaning an obligation with the index 0 exists, the witnessing path, which consists of a series of states, can be extracted from the obligations by following the chain of parent obligation references and storing the corresponding states in a list. A witnessing path of length $n$ consists of the states $s_0, \dots, s_{n-1}$. The witnessing path can then be converted to a plan by iterating over every two consecutive states $(s_{i-1}, s_i)$ in the witnessing path and storing the operator $o_i \in \mathcal{O}$ where $s_{i-1} \models \mathit{pre}(o_i)$ and $s_{i-1}[\![o_i]\!] = s_i$. In case multiple operators meet those two criteria, any one of those operators can be selected.

## 3.2 The Algorithm

As mentioned above, PDR is structured into two separate phases, the path construction phase, and the clause propagation phase. In every iteration step $k$, the path construction phase is executed first, followed by the clause propagation phase. The pseudo-code for the implementation of PDR can be found in Algorithm 1.

**Initialization** Before the main iteration, the layer $L_0$ is initialized with the goal formula. The goal formula of a positive STRIPS task is always a cube, which is a conjunction of literals. Therefore, the conversion to CNF is done by iterating over each literal in the goal formula and treating it as a unit-clause $c$. This clause is then inserted into the CNF formula. All other layers $L_i$ with $i > 0$ are initialized with the empty set. Next, the main iteration starts, counted by the variable $k$ starting with the value 0 and only aborting if either a plan has been found, or the conclusion that no plan exists has been reached.

**Path Construction Phase** The goal of the path construction phase is to build a witnessing path from the initial state to a goal state. If such a witnessing path exists, it will be found during the path construction phase, and the algorithm terminates. Otherwise, if no witnessing path of such length exists, the path construction phase concludes.

The path construction phase starts by checking if the initial state $s_I$ is a model of the current layer $L_k$, and only if this is the case, the rest of the path construction phase, as illustrated in Figure 2, is executed. Otherwise if $s_I \not\models L_k$, the initial state is not in layer $L_k$ which means there is no path from the initial state to a goal state with a length of $k$ or less. Next on line 8, the obligation queue $Q$ is initialized and the obligation $q = (s_I, *, k)$ is pushed into it. With this, we store the fact that the initial state $s_I$ is the tip of the zero-length path starting at $s_I$. This corresponds to the step visualized in Figure 2a, where the newly inserted obligation is $q_1$.

On line 10, a loop is started that terminates when the priority queue has no more elements. The first step in this loop is to pop an obligation $q = (s, p, i)$ from the obligation queue with the smallest index $i$. If $i$ is zero, $s \models L_0$ and $s$ is therefore a goal state, meaning that the search has concluded. The plan is extracted from the obligation and returned. This corresponds to the step visualized in Figure 2e. Otherwise, the *extend* function is called with the state of the obligation and with the layer $L_{i-1}$ as arguments. The *extend* function returns either a successor $t$ of $s$, such that $t$ is in the layer $L_{i-1}$, or a reason $r \subseteq Lits(s)$, which is a cube containing the literals of $s$ that prevent any successor of $s$ from being in $L_{i-1}$. A detailed explanation of the *extend* function follows in Section 3.3. If the successor is returned, the popped obligation $q$ is pushed back into the queue, along with the new obligation $(t, q, i-1)$, and the loop is continued. Otherwise, the negation of the reason $r$ is inserted into all layers $L_j$ with $j \leq i$, which strictly strengthens the layers. The reason $r$ removes at least the state $s$ from any of the updated layers. This prevents further path construction from state $s$ for any layer $L_j$ with $0 \leq j \leq i$. Additionally, on line 24 if the popped obligation has an index of less than $k$, it is pushed back into the obligation with a higher index than before. This obligation rescheduling is not necessary for the correctness of PDR, but allows it to find non-optimal length

**Algorithm 1** PDR Planning Algorithm

**Input** A positive unit cost STRIPS planning task $\Pi = (\mathcal{V}, \mathcal{O}, s_I, s_*)$.

**Output** A plan for $\Pi$ or "unsolvable" if no plan exists for $\Pi$.

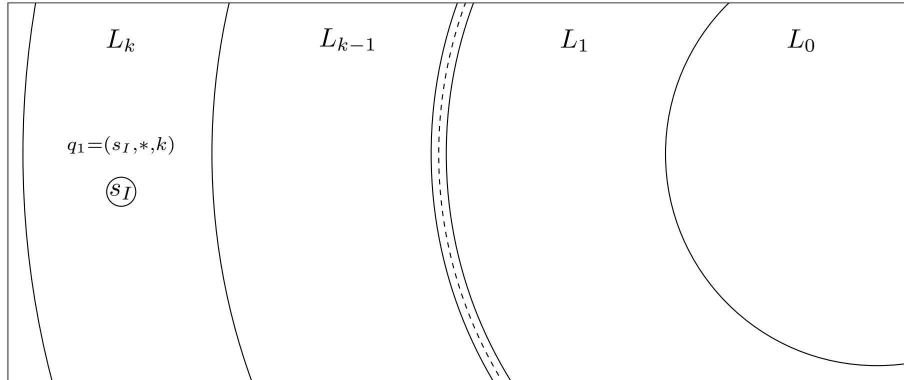
1: $\triangleright$ Initializing the layer $L_0$ as a set of unit clauses from the goal cube. $\triangleleft$
2: $L_0 \leftarrow \{\{c\} \mid c \in s_*\}$
3: **for** $j > 0$ **do**
4: $\quad L_j \leftarrow \emptyset$;
5: **end for**
6: **for** $k = 0, 1, \ldots$ **do** $\triangleright$ Start of the path construction phase
7: $\quad$ **if** $s_I \models L_k$ **then**
8: $\quad\quad Q \leftarrow$ min_priority_queue()
9: $\quad\quad Q$.push($(s_I, *, k)$)
10: $\quad\quad$ **while not** $Q$.empty() **do**
11: $\quad\quad\quad q' \leftarrow Q$.pop()
12: $\quad\quad\quad (s, p, i) \leftarrow q'$
13: $\quad\quad\quad$ **if** $i = 0$ **then**
14: $\quad\quad\quad\quad$ **return** extract_plan($(s, p, i)$)
15: $\quad\quad\quad$ **end if**
16: $\quad\quad\quad (\mathit{success}, x) \leftarrow$ **extend**$(s, L_{i-1})$
17: $\quad\quad\quad$ **if** *success* **then** $\triangleright$ $x$ is a successor state
18: $\quad\quad\quad\quad Q$.push($(x, q', i-1)$)
19: $\quad\quad\quad\quad Q$.push($(s, p, i)$)
20: $\quad\quad\quad$ **else** $\triangleright$ $x$ is a reason $x \subseteq \mathit{Lits}(s)$
21: $\quad\quad\quad\quad$ **for** $j = 0, \ldots, i$ **do**
22: $\quad\quad\quad\quad\quad L_j \leftarrow L_j \cup \{\neg x\}$
23: $\quad\quad\quad\quad$ **end for**
24: $\quad\quad\quad\quad$ **if** $i < k$ **then** $\triangleright$ Obligation rescheduling
25: $\quad\quad\quad\quad\quad Q$.push($(s, p, i+1)$)
26: $\quad\quad\quad\quad$ **end if**
27: $\quad\quad\quad$ **end if**
28: $\quad\quad$ **end while**
29: $\quad$ **end if**
30: $\quad$ **for** $i = 1, \ldots, k+1$ **do** $\triangleright$ Clause propagation
31: $\quad\quad$ **for** $c \in L_{i-1} \setminus L_i$ **do**
32: $\quad\quad\quad s_c \leftarrow \{p \to \bot \mid p \in c\} \cup \{p \to \top \mid p \in X \setminus c\}$
33: $\quad\quad\quad$ **if** all $o \in \mathcal{O} : s_c \not\models \mathit{pre}(o)$ or $s_c[\![o]\!] \not\models L_{i-1}$ **then**
34: $\quad\quad\quad\quad L_i \leftarrow L_i \cup \{c\}$
35: $\quad\quad\quad$ **end if**
36: $\quad\quad$ **end for**
37: $\quad\quad$ **if** $L_{i-1} = L_i$ **then**
38: $\quad\quad\quad$ **return** "unsolvable" $\triangleright$ No plan possible
39: $\quad\quad$ **end if**
40: $\quad$ **end for**
41: **end for**

**Algorithm 2** Function extend$(s, L)$

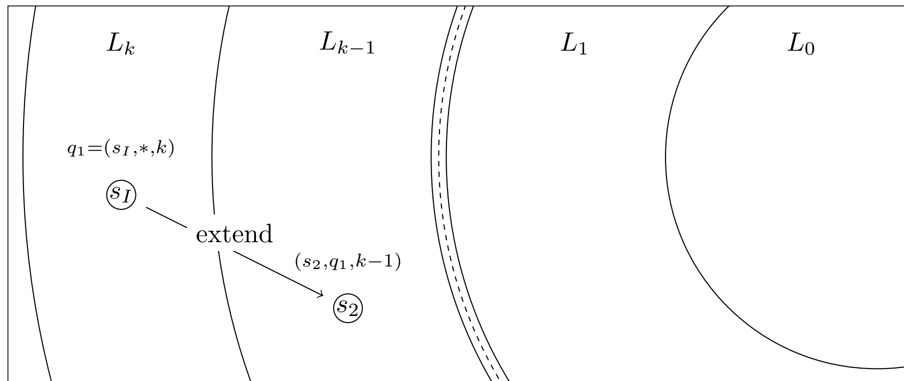
**Input** A state $s$, and a layer $L$.

**Output** A tuple $(success, x)$, with $success$ being a truth value and $x$ being a cube. If $success = \textbf{true}$, then $x = Lits(t)$ where $t$ is a successor of $s$ such that $t \models L$. Otherwise, $success = \textbf{false}$ and $x \subseteq Lits(s)$ is a reason such that no state satisfying $x$ has a successor satisfying $L$.

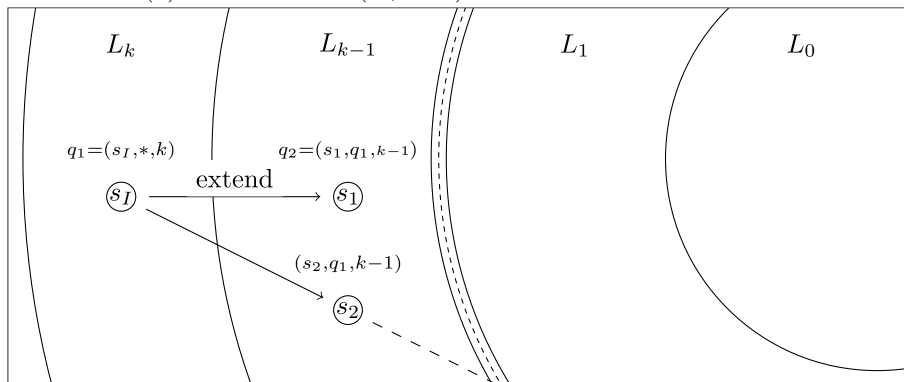
1: **assert** $s \not\models L$
2: $L^s \leftarrow \{c \in L \mid s \not\models c\}$
3: $R_{noop} \leftarrow \{\neg c \mid c \in L^s\}$
4: **assert** $R_{noop} \neq \emptyset$
5: $\mathcal{R} \leftarrow \{R_{noop}\}$
6: **for** $o \in \mathcal{O}$ **do**
7: $\quad pre^s_o \leftarrow \{l \in pre(o) \mid s \not\models l\}$
8: $\quad t \leftarrow s[\![(\emptyset, eff(o))]\!]$ ▷ apply $o$ regradless of the precondition
9: $\quad L^t \leftarrow \{c \in L \mid t \not\models c\}$
10: $\quad$**if** $pre^s_o = \emptyset$ and $L^t = \emptyset$ **then**
11: $\quad\quad$**return** $(\textbf{successor}, t)$
12: $\quad$**else if** $L^s \subseteq L^t$ **then**
13: $\quad\quad$**continue**
14: $\quad$**else**
15: $\quad\quad L^t_0 \leftarrow \{c \in L^t \mid c \cap pre^s_o = \emptyset\}$
16: $\quad\quad R_o \leftarrow \{\{\neg l\} \mid l \in pre^s_o\} \cup \{\{\neg l \mid l \in c \text{ and } \neg l \notin eff(o)\} \mid c \in L^t_0\}$
17: $\quad\quad \mathcal{R} \leftarrow \mathcal{R} \cup \{R_o\}$
18: $\quad$**end if**
19: **end for**
20: $r \leftarrow \emptyset$
21: **for** $R_o \in \mathcal{R}$ ordered by $|R_o|$ from small to large **do**
22: $\quad r_o \leftarrow \min_{r_o \in R_o}\{|r \cup r_o|\}$ ▷ find an $r_o$ that increases $|r|$ the least
23: $\quad r \leftarrow r \cup r_o$
24: **end for**
25: **return** $(\textbf{reason}, r)$



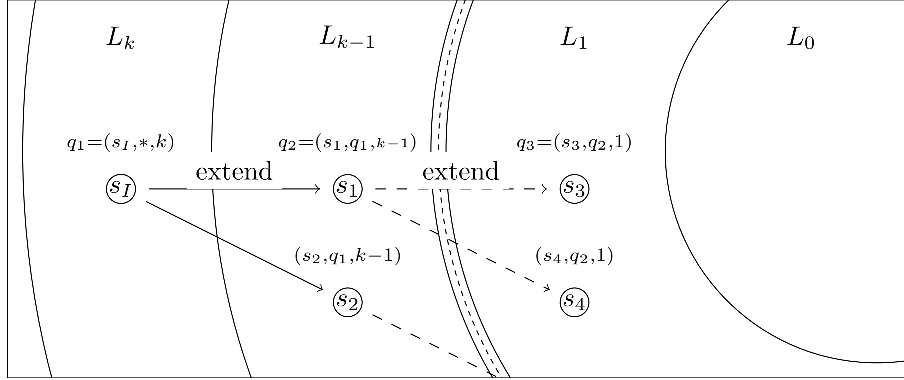

(a) Iteration step $i = 1$ of the path construction phase. The only existing obligation is $q_1$ containing the initial state $s_I$ in layer $L_k$.



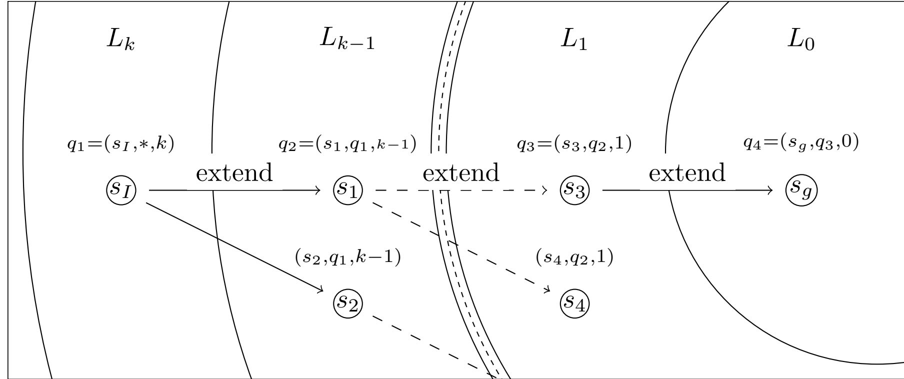

(b) A call to $extend(s_I, L_{k-1})$ returned a successor state $s_2$.




(c) Calls to $extend(s_2, L_{k-2})$ resulted in a new state, a further call to *extend* from this state returns a reason, and no path to layer $L_0$ is found. A call to $extend(s_I, L_{k-1})$ resulted in the new state $s_1$.




(d) Calls to the *extend* function resulted in two paths of obligations ending in the states $s_3$ and $s_4$ in the layer $L_1$. A further call to $extend(s_4, L_0)$ returns a reason.




(e) A call $extend(s_3, L_0)$ resulted in the state $s_g$. The corresponding obligation $q_4$ has a priority value of 0. The algorithm terminates in the next step. The witnessing path is extracted from the obligation $q_4$ by following the parent obligation references $q_4 \to q_3 \to \cdots \to q_2 \to q_1$. The witnessing path $\langle s_I, s_1, \ldots, s_3, s_g \rangle$ is returned.

Figure 2: Illustration of the path construction phase. Shown are the states $s_I, s_1, \ldots, s_4, s_g$ and the obligations $q_1, \ldots q_4$. The dashed layer line between layer $L_{k-1}$ and $L_1$ represents an arbitrary number of layers, and the dashed arrows across those layers represent an arbitrary number of expansion steps. Each sub-figure shows the state of the path construction phase at a progressive iteration step. The strengthening of the layers through reasons returned by the extend function is not shown.

paths, while typically increasing the performance of the algorithm. Without this obligation rescheduling, the obligation $q = (s, p, i)$ will be discarded, and work continues on the obligation $p$. This corresponds to the strict backtracking behavior with the obligation queue $Q$ acting as the stack.

**Clause Propagation Phase** After the construction phase has concluded for an iteration step $k$, the clause propagation phase starts. Clause propagation has the goal to copy clauses from low-index layers to high-index layers, and therefore strengthening the high index layers. This is done by iterating over $i = 1, \dots, k+1$ and checking for every clause $c \in L_{i-1} \setminus L_i$ if it could be pushed into the layer $L_i$. For a clause to be pushed into a higher layer without violating any of the layer invariants, the formula

$$F_c \coloneqq \neg c \wedge T \wedge (L_{i-1})'$$

must be unsatisfiable. To express this requirement without a SAT solver, consider the positive STRIPS task $\Pi = (\mathcal{V}, \mathcal{O}, s_I, s_*)$ with the transition formula $T$, $L$ a set of positive clauses over $\mathcal{V}$ and $c$ a positive clause over $\mathcal{V}$. Additionally, $s_c$ is an assignment for every $p \in \mathcal{V}$ with

$$s_c(p) = \begin{cases} \bot & \text{if } p \in c \\ \top & \text{otherwise.} \end{cases}$$

Suda shows in Lemma 4 that [2]

> $F_c$ is satisfiable if and only if there is an operator $o \in \mathcal{O}$ such that $s_c \models \mathit{pre}(o)$ and $s_c[\![o]\!] \models L$.

Therefore, for $F_c$ to be unsatisfiable for layer $L_{i-1}$, every operator $o \in \mathcal{O}$ must satisfy either $s_c \not\models \mathit{pre}(o)$ or $s_c[\![o]\!] \not\models L_{i-1}$, or both. If this is the case, the clause $c$ can be added to layer $L_i$ without violating the layer invariants.

After every iteration of the clause propagation phase, we check for the equality of neighboring layers. If indeed two equal layers $L_{i-1}$ and $L_i$ with $0 < i \leq k+1$ are found, the algorithm is terminated, returning "unsolvable". In this case, no path of length $k$ or longer can exist. Suda proves this formally as part of Theorem 1, however, this can also be explained by intuition: Consider a case where the layers $L_{i-1}$ and $L_i$ with $0 < i \leq k+1$ are equal. Assume there exists witnessing path $\pi = (s_k = s_I, \dots, s_i, s_{i-1}, \dots, s_0 = s_*)$. Any state $s_j$ with $j \in \{1, \dots, k\}$ fulfills the conditions $s_j \models L_j$ and $s_j \not\models L_{j-1}$. The second condition holds because the layer $L_{j-1}$ would have been strengthened during the path construction phase if this was not the case before. We can now follow from $L_{i-1} = L_i$ that $s_i \models L_i$ and $s_i \not\models L_i$, which is a contradiction.

## 3.3 Extending the Witnessing Path

This section will describe the $\mathit{extend}(s, L)$ function, as used in the path construction phase, in detail. The goal of the function $\mathit{extend}(s, L)$, with $s$ being a state and $L$ being a layer, is to provide a successor state $t$ of $s$ that is a model of

[2] The notation of this lemma has been adapted. The action $a$ has been renamed to the operator $o$, and the notation for the successor state of $s_c$ applying $a$ has been written as $s_c[\![o]\!]$ instead of $\mathit{apply}(s_c, a)$.

$L$, if such a $t$ exists. This case is called a successful path extension. If no such $t$ exists, *extend* provides a reason $r$, which is a cube and a subset of the literals of $Lits(s)$, such that no state that satisfies $r$ has a successor that satisfies $L$.

The reason should be preferably small, meaning it should contain as few literals as possible. Small reasons are preferred because they are used to strengthen the layers. Larger layers, in terms of the number of clauses and the number of literals inside of those clauses, impact the performance of PDR negatively. This performance impact happens both in terms of run-time complexity when computing operations such as the satisfies relation, and also because larger layer clauses lead to the layer excluding less states, which potentially leads to more iterations in the path construction phase.

The full pseudo code for the *extend* function can be seen in Algorithm 2. The *extend* function works by iterating over all of the operators $o \in \mathcal{O}$ and generating a successor state $t_o$ if $o$ is applicable. If such a successor $t_o$ is a model of $L$, the path extension is successful and the successor is returned. In the unsuccessful case, a reason has to be calculated.

The idea of the unsuccessful case is to collect a reason set $R_o$ for every operator $o$. The overall full reason $r_f$ is the union

$$r_f = \bigcup_{o \in \mathcal{O}} R_o.$$

Suda shows how it is possible to construct a smaller, but still valid reason, by selecting reason contribution $r_o \in R_o$ such that the union

$$r = \bigcup_{o \in \mathcal{O}} r_o \tag{1}$$

results in a small $|r|$, while still satisfying the condition that no state that satisfies $r$ has a successor that satisfies $L$. Suda calls this method "minimizing the reason".

**Constructing the Set of Reasons** The set of reasons $R_o$ for an operator $o \in \mathcal{O}$ is comprised of reasons acquired as follows. If the operator $o$ is not applicable for the state $s$, then there is at least one literal $l \in pre(o)$ that is false in $s$. The negation of $l$ is one such reason $\{\neg l\} \subseteq Lits(s)$ that we add to $R_o$.

The next step is to compute the successor state $t_o = s[\![o']\!]$, where the operator $o'$ consists of $pre(o') = \top$ and $eff(o') = eff(o)$. If this successor state $t_o$ is not a model of $L$, there is at least one clause $c \in L$ where $t_o \not\models c$. This is the case when either $s$ is not a model of $c$ due to some literals which are not changed by $eff(o)$, or $s$ is a model of $c$, but $eff(o)$ introduces literals that prevent $t_o$ from being a model of $c$. For every such clause, we add the reason $r_c$ to $R_o$, which consists of the negations of the literals $l \in c$. To reduce the size of the reason $r_c$, we can exclude the literals $\neg l \in eff(o)$. This still results in a valid reason because as long as $s$ satisfies $r_c$, the successor $t_o$ cannot satisfy $c$. Formally, the set of reasons is defined as on line 16.

**Reason Subsumption** When the extend function returns a reason $r$, we want this reason to have the smallest size $|r|$ as possible. Smaller reasons are preferable as mentioned above, because the size of the layers influences the performance of the PDR algorithm.

Before computing the overall reason $r$ from the sets $R_o$, there are subsumptions between individual reasons $r_o \in R_o$, as well as between the reason sets $R_o$. For individual reasons $r_1, r_2 \in R_o$, the subsumption is the subset relation. If $r_1 \subseteq r_2$, it is not necessary to keep both inside $R_o$, since keeping just the smaller cube $r_1$ is sufficient. Keeping $r_1$ leads to the result of the union in Equation 1 being smaller.

Additionally to the subsumption of individual reasons, a reason set $R_o$ for an operator $o$ is subsumed by a reason set $R_u$ for an operator $u$ if for every $r_u \in R_u$ there exists an $r_o \in R_o$ such that $r_o \subseteq r_u$. In this case, we can discard the whole reason set $R_o$. As mentioned by Suda, this is only applied in regard to the $R_{noop}$ reason set corresponding to the (artificially inserted) no-op operator due to performance reasons. The no-op action is included to ensure the correctness of the PDR algorithm, and the $R_{noop}$ reason set contains the reasons why $s$ is not a model of $L$. The reason set $R_{noop}$ always contains at least one cube, since the extend function is only called for $s \not\models L$. If an operator $o$ does not result in any of these clauses being made true, the resulting reason set $R_o$ is subsumed by $R_{noop}$ and will be skipped. This is implemented on line 12.

**The Overall Reason** The whole reason $r$ is computed by selecting a reason $r_o \in R_o$ for every operator $o$ such that the union from Equation 1 is as small as possible. Selecting the right reasons $r_o$ is difficult. Suda even argues that finding the optimal solution is NP-complete. Instead of trying to find an optimal solution, we use a greedy approach as implemented on lines 20 - 23.

First, the variable of the overall reason $r$ is initialized with an empty set. We iterate over all $R_o$, ordered by increasing $|R_o|$. We then select an element $r_o \in R_o$ such that $|r \cup r_o|$ is as small as possible, which is then added to the overall reason $r$. After the iteration, the overall reason $r$ is returned from the extend function.

The greedy minimization of $r$ happens due to the ordering of $R_o$. For $R_{o_1}$ containing a single cube $c$, we don't have any freedom to choose the contribution to the overall reason: $c$ is the only reason that can be contributed. Contrary to this, for a large $R_{o_2} = \{c_1, c_2, \dots\}$ we have the freedom to choose any one of those cubes. By ordering $R_o$ according to their size, the choice of which cube $c \in R_o$ to contribute to the overall reason is better informed by the previously contributed cubes.

Suda proposes an optional additional reason minimization step to further reduce the size of the reason, which is not explicitly covered here.

# 4 Heuristics in PDR

As mentioned in Chapter 1, the goal of this thesis is to explore ways to pre-populate – or seed – the layers before they are accessed for the first time by the PDR algorithm. We expect this to lead to performance improvements of the whole algorithm, at the cost of higher memory consumption, depending on how much the layers are seeded. In this section we will explore how to seed the layer with a generic heuristic function, and then how to improve this with a specialized implementation using the pattern database heuristic.

## 4.1 Seeding Layers

Recall from Section 3.1 that layers are formulas in conjunctive normal form. A layer $L_i$ represents a super-set of all states that have a goal distance of at most $i$. PDR iteratively refines the layers to progressively exclude states that are too far from the goal. The goal of seeding the layers is to initialize each layer with a formula that excludes as many states as possible while making sure the layer invariants still hold. By excluding states from a layer $L_i$, the PDR algorithm is more likely to pick a state $s \models L_i$ in the path construction phase that is actually at most $i$ steps away from the goal. We introduce the notion of a perfect layer $L_i^*$, that represents exactly the set of states with a goal distance of $i$ or less. For such a perfect layer, the algorithm is guaranteed to pick a state $s_i$ that is at most $i$ steps away from the goal. By intuition, using a fast heuristic function to remove some of the work from the algorithm should lead to higher overall performance.

**Theorem 1.** *Given the perfect layers $L_i^*$ and $L_{i-1}^*$, and a state $s$ with $s \models L_i^*$ and $s \not\models L_{i-1}^*$, there must be a successor $t$ of $s$ with $t \models L_{i-1}^*$, as shown in Figure 3.*

*Proof.* Given a state $s$ that models the clauses of the layer $L_i^*$ and not of the layer $L_{i-1}^*$. Assume that for the set of all successors $T$ of $s$, no successor $t \in T$ models the clauses of $L_{i-1}^*$. This implies that all successors of $s$ are more than $i-1$ steps away from the goal, and therefore $s$ is at least $i+1$, not $i$ steps away from the goal. This contradicts the definition of the layer $L_i^*$, which declares that all states that are a model of it, are at most $i$ steps away from the goal. Therefore, at least one successor state $t \in T$ of $s$ with $t \models L_{i-1}^*$ exists. □

The function call $extend(s, L_{i-1})$ is guaranteed to return a successor of $s$ that satisfies $L_{i-1}$, if such a state exists. From this and from Theorem 1 it follows that the path construction phase always successfully finds a successor $t$, and therefore never needs to strengthen a layer.

Through seeding, we strengthen each layer $L_i$. This reduces the likelihood of expanding an obligation with the state $s$ for a layer $L_i$ in the path construction phase that is more than $i$ steps away from a goal. This leads to fewer obligation expansions and therefore fewer iterations in the path construction phase. By seeding the layers with as restrictive formulas as possible, while making sure they still fulfill the invariants, the PDR algorithm should therefore amortize the pre-computation overhead and find a solution faster.

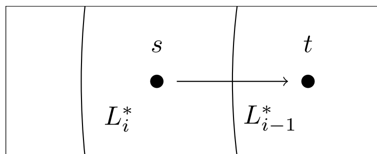



Figure 3: An illustration of the perfect layers $L_i^*$ and $L_{i-1}^*$, as well as a state $s$ and its successor state $t$.

## 4.2 Layers from Heuristics

By intuition, using a fast heuristic function to remove some of the work from the algorithm should lead to higher overall performance.

Seeding the layers requires insight into the search space of the particular problem the algorithm is trying to solve. The seeding algorithm needs a way to classify states for each layer $L_i$ into the categories "at least $i+1$ steps away from the goal" and "possibly fewer than $i+1$ steps away from the goal". The categories can be this loose because PDR explicitly uses layers that are a superset of states that are at most $i$ steps away from the goal. It is therefore enough to find an estimate of the distance to the goal that is guaranteed to be optimistic, meaning that for a state that is $j$ steps away from the goal, the estimate must be at most $j$. As mentioned in Section 2.4, heuristic functions are designed to find such estimations. Specifically, admissible heuristics fulfill this requirement.

To seed a layer $L_i$ we can find the set of states $S^{h,i} := \{s \in S \mid h(s) > i\}$ where each element is a state with the heuristic value greater than $i$. Since we require the heuristic function to be admissible, each state in $S^{h,i}$ is guaranteed to be at least $i+1$ steps away from the goal. To strengthen the layer $L_i$, we add a new clause $c$ to the layer for every state $s \in S^{h,i}$, where the clause is of the form $c = \neg(\mathit{Lits}(s))$. This formula ensures that all states in $S^{h,i}$ are not a model of the seeded layer $L_i$.

## 4.3 Layers from Pattern Database Heuristics

In practice state spaces tend to be very large, and adding a clause to a layer for every state is generally not feasible. Instead, it makes sense to take advantage of the structure of the state space, not just the goal distance of every state. It is preferable to exclude a whole class of states from the layer at once with a single clause. For example, consider a state space where all states with the variable $v \in \mathcal{V}$ assigned to $\perp$ are more than $i$ steps away from the goal:

$$h(S_{\{(v \to \perp)\}}) > i.$$

It is not necessary to add a clause for every state in $S_{\{(v\to\perp)\}}$. Instead, we should add the single clause $(v) = (\neg\mathit{Lits}(\{v \to \perp\}))$. Adding this clause to the layer $L_i$, has the same effect as adding all the clauses $\{(\neg\mathit{Lits}(s)) \mid s \in S_{\{v\to\perp\}}\}$, but the formula is much more compact.

As mentioned in Section 2.4.1, the pattern database heuristic makes use of the structure of the state space through the projection pattern $P$. It is possible to exploit the pattern to exclude a whole set of states in the same way as demonstrated above. However, instead of only considering a single variable,

we consider all variables in the pattern $P$. We exclude all projected states from the layer that are more than $i$ steps away from the goal. The first step is enumerating all projected states $S_P$ and calculating their distance from the goal. Conveniently, this is already done by the pattern database heuristic in the pre-computation step. Next, the states are filtered to extract all projected states that are more than $i$ steps away from the goal: $S_P^{h,i} := \{h(s) > i \mid s \in S_P\}$. Each state in $S_P^{h,i}$ is potentially the projection of a high number of states, depending on the pattern. To exclude the projected states, and therefore also their corresponding non-projected states from the layer we insert a clause for each projected state

$$L'_i = L_i \wedge \left( \bigwedge_{\{s \in S_P^{h,i}\}} \neg \left( Lits(s) \right) \right). \tag{2}$$

As mentioned in Chapter 3, the PDR implementation without SAT-solver requires the STRIPS task to be positive. Due to this, the layers must consist of only positive clauses for the implementation to be correct. However, by seeding the layers as described above, the seeded layers are not guaranteed to only contain positive clauses. The following is an example that leads to a non-positive clause:

Assume we want to seed a layer $L_i$, and the projected state $\pi_P(s) = \{a \to \top, b \to \bot\})$ has a goal distance greater than $i$. The clause for this projected state is $c = (\neg Lits(\pi_P(s)) = (\neg a \vee b)$. This clause is non-positive.

Additionally, recall from Section 2.4.1, pattern databases are usually defined for tasks in finite domain representation, as opposed to STRIPS tasks. We will assume the planning tasks are in FDR for the rest of this section.

Recall from Section 2.3.1, a task in FDR has variables $\hat{v} \in \hat{\mathcal{V}}$ and each variable $\hat{v}$ has domain $D_{\hat{v}}$. The equivalent planning task converted back to a STRIPS representation defines the variables $\mathcal{V} = \{(\hat{v}, d) \mid \hat{v} \in \hat{\mathcal{V}} \text{ and } d \in D_{\hat{v}}\}$. In any single variable assignment in the FDR task, a variable can only be assigned to one value in $D_v$.

As mentioned previously, the PDR algorithm expects the layers to consist of only positive clauses. However, trying to exclude the FDR state $s = \{(v \to A)\}$, with the domain $D_v = \{A, B, C\}$, from a layer results in the clause $\{\neg(v, A), (v, B), (v, C)\}$, which contains the negative literal $\neg(v, A)$. To prevent this, we can exploit that $s$ can also be represented as the cube

$$Lits(s) \equiv \bigcup_{d' \in D_v, d' \neq A} \neg d = \{\neg(v, B), \neg(v, C)\}.$$

When considering the states reachable from the initial state, this representation is equivalent to $Lits(s)$ because both cubes $\{(v, A)\}$ and $\{\neg(v, B), \neg(v, C)\}$ imply each other due to the mutual exclusivity of the literals. By applying this procedure for every variable $v \in \hat{\mathcal{V}}$ of the FDR task, every projected state can be represented using a negative cube. Excluding this state from a layer is achieved by negating this cube, yielding a positive clause that is inserted into the layer.

Even though this results in layers that include potentially unreachable states, this does not violate any of the layer invariants as defined in Chapter 3 as shown by Theorem 4.

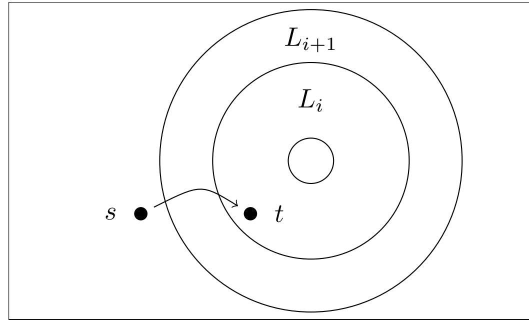



Figure 4: An example of a scenario that violates **Layer Invariant 3**. Shown are the states $s$ and its successor $t$ and the layers $L_i$ and $L_{i+1}$, where $t \models L_i$ and $s \not\models L_{i+1}$.

**Theorem 2.** *For any two seeded layers $L_i$ and $L_j$ with $i > j$, $L_i$ is a subset of $L_j$.*

*Proof.* Assume $P$ is a pattern in $\Sigma$, and $S_P^{h,i}$ is the set of projected states with a goal distance of more than $i$ steps. From $i > j$ follows that $S_P^{h,i} \subseteq S_P^{h,j}$, since all projected states with a goal distance of more than $i$ also have a goal distance of more than $j$. From the definition of Equation 2 it follows that $L_i \subseteq L_j$. $\square$

**Theorem 3.** *Layers seeded with the PDB heuristic do not violate the layer invariant 3.*

*Proof.* The layer invariant 3 $(L_i)' \wedge T \Rightarrow L_{i+1}$ states, that all predecessors of the states in layer $L_i$ are in layer $L_{i+1}$. By applying the contradiction theorem it follows that the statement $(L_i)' \wedge T \wedge \neg L_{i+1}$ must be unsatisfiable. This means, that a situation where a state is in $L_i$ and its predecessor is not in $L_{i+1}$ is impossible. We will prove the invariant by contradiction: Consider the states $s$ and $t$ where $t$ is a successor of $s$. Assume that $t \models L_i$ and $s \not\models L_{i+1}$ where $L_i$ and $L_{i+1}$ are seeded layers. Figure 4 shows an example of such a scenario. From $t \models L_i$ we know that $h_{pdb}(t) \leq i$, and from $s \not\models L_{i+1}$ we know that $h_{pdb}(s) > i+1$. This follows directly from the seeding of the layers. Every layer $L_j$ only models states $s_j$ that have a heuristic value $h_{pdb}(s_j) \leq j$. The pattern database heuristic is consistent] [6], meaning that $h_{pdb}(s) \leq h_{pdb}(t)+1$ since the planning task has unit costs. From this, we can follow that $i + 1 < h_{pdb}(s) \leq h_{pdb}(t) + 1 \leq i + 1$ and therefore $i + 1 < i + 1$ which is a contradiction. $\square$

**Theorem 4.** *Layers seeded with the PDB heuristic do not violate any of the layer invariants.*

*Proof.* Recall the layer invariants from Chapter 3:

1. $L_0$ represents only the goal states.
2. $L_{j+1} \subseteq L_j$.
3. $(L_j)' \wedge T \Rightarrow L_{j+1}$ for any $j \geq 0$.

**Invariant 1:** By the definition of the PDR algorithm, the layer $L_0$ is initialized (before seeding) to be equivalent to the goal formula. The seeding step strengthens the layer by adding additional clauses to it. The clauses that are added during the seeding step are the negation of $Lits(s)$ of abstract states $s$ with a heuristic value of more than zero. The PDB heuristic only has non-zero heuristic values for abstract states with at least one variable $v$ that is not assigned to its goal value $g$. Therefore only states with a goal distance of more than zero are excluded from the layer. Since the only states with a goal distance of zero are goal states, the layer $L_0$ before and after seeding is equivalent. **Invariant 2** follows directly from Theorem 2. **Invariant 3** holds as shown in Theorem 3. □

With this, we have found a way to seed the layers with a pattern database heuristic, and shown that the resulting seeded layers do not violate the layer invariants. We can conclude that the PDR algorithm with seeding using the proposed method is correct.

# 5 Implementation

This section describes the implementation of the PDR algorithm as a search engine in the Fast Downward [9] planning system, as well as the layer seeding with the pattern database heuristic.

## 5.1 PDR in the Fast Downward Planning System

The PDR algorithm is implemented as a distinct search engine in Fast Downward[3]. Specifically, the search-engine interface is satisfied by implementing the methods `initialize` and `step`.

The initialization step is not absolutely necessary for the implementation of PDR, since no tasks have to be completed before the main loop besides initializing the layers. Since the number of layers can not be determined beforehand, we only initialize the layer $L_0$ here. Additionally, the effects of all actions are fetched from Fast Downward and cached for later use, as part of an optional optimization.

The `step` method returns a value that indicates the current state of the search, such as `in_progress`, `solved`, or `failed`. The method is repeatedly called by Fast Downward until it returns one of the values that indicate that the search task has concluded. This maps directly to the main iteration step of the PDR algorithm. Each call of the `step` function corresponds to an iteration of the outermost loop in the PDR algorithm.

### 5.1.1 Data Structures

The PDR algorithm as described in Chapter 3 consists mostly of set operations. While cubes, clauses, and formulas in CNF are just represented as sets in the pseudo-code, using raw set objects for the implementation is ill-suited. While reasoning about a set that is called a cube is easy, a set does not intrinsically have the property that it is a cube. To add such kind of metadata, we create the following specialized data structures:

**Literal** A literal consists, as defined in Section 2.1, out of a variable and an indicator if it is positive or negative. Since Fast Downward operates in FDR, the domain $D_v$ of a variable $v$ can contain more than the values $\top$ and $\bot$. Since the PDR algorithm can not operate on FDR literals, we convert them to STRIPS literals as described in Section 2.3.1. We end up with a data structure containing the variable and the FDR value as integers and the indicator if the value is positive or negative as a boolean. For debugging reasons, a reference to the Fast Downward type `FactProxy` is stored as well, which holds the original name of the literal in the planning task. As a memory usage optimization, this reference could be dropped, as no code in the PDR implementation depends on it.

**LiteralSet** All formulas used in the PDR algorithm are either clauses, cubes, or formulas in CNF or DNF. We model clauses and cubes as sets of literals with an additional indicator if the set is to be treated as a clause or a cube. To

[3] The implementation can be found at https://github.com/Tiim/fast-downward-pdr

store the literals, the `LiteralSet` contains a `literals` member variable of the container type `std::unordered_set` which stores a set of `Literal` values. The additional indicator if the set is to be considered a clause of a cube is stored as an enum value `SetType::CLAUSE`, or `SetType::CUBE`. The `LiteralSet` class implements common set operations used by the PDR algorithm, such as: `set_union` which implements the union set operator "∪", `set_minus` which implements the set operator "\", as well as the `models` function, which accepts a layer $L$ and returns true if the `LiteralSet` $c$ fulfills $c \models L$. Some other member functions are implemented as well which are not mentioned here.

**SetOfLiteralSets** As mentioned above, formulas used in PDR can also be in CNF or DNF. Both of those normal forms can be represented as either sets of clauses or sets of cubes respectively. To store the clauses or cubes, the `SetOfLiteralSets` contains a `sets` member variable of the container type `std::unordered_set` which stores a set of `LiteralSet` values. An additional indicator for the type of normal form is not strictly needed, since the elements of the set already contain this information. It is however still stored for explicitness when the set is empty, and to ensure correctness in the case a clause was inserted into a DNF or a cube was inserted into a CNF. While this should never happen, such an error would be hard to find without such an indicator and additional checks on every insert. The `SetOfLiteralSets` class also implements some convenience methods that operate on the `sets` member variable, which are not mentioned here.

**Layer** A special case of a formula in CNF is the data structure of a layer. Since most functionality is the same it is implemented the same way as the `SetOfLiteralSets` class. The main difference in the internal representation is the delta-encoding of the `sets` member variable. As described in Section 3.1, layer invariant 2 states that $L_{j+1} \subseteq L_j$ for every $j > 0$. For this reason, and since the layers are strictly ordered, it is possible for each layer $L_i$ to only store the layer delta $L_i^\Delta = L_i \setminus L_{i+1}$, together with a reference to the parent layer $L_{i-1}$ and the child layer $L_{i+1}$, provided they exist and have been initialized. The layer delta $L_i^\Delta$ contains the clauses that are in all layers up to the layer $L_i$ and are not members of any following layers. The union of all child layer deltas results in the layer

$$L_i = \bigcup_{i \leq j} L_j^\Delta.$$

Conveniently, since each layer saves its delta encoding, the set operation $L_{i-1} \setminus L_i$, as required in the clause propagation phase, directly evaluates to $L_{i-1}^\Delta$ and does not need to be computed.

**State** As defined in Section 2.3, a state is a full assignment. Since we already have a way to convert assignments to cubes we can conveniently represent a state $s$ as the cube $Lits(s)$ using the `LiteralSet` class.

**Obligation** As described in Section 3.1, an obligation is defined as a tuple $q = (s, q', p)$. The elements of the tuple can directly be mapped to the member variables of the `Obligation` class: The `state` variable is a `LiteralSet` with a set type of `SetType::CUBE` as mentioned above. The priority $p$ is a simple

integer, and the reference to the parent obligation is a member variable of type `std::shared_ptr<Obligation>`. The obligation $(s_I, *, k)$ has a parent value of `nullptr` to signal that no parent obligation exists.

## 5.2 Heuristic Seeding of Layers

The PDR search engine instance holds a list of all accessed layers. Whenever a layer $L_k$ needs to be accessed during the run-time of the PDR algorithm, the `get_layer` method of the search engine object is invoked. If this layer has been accessed before, it is returned from the layer list directly. Otherwise, it is initialized as an empty `Layer` object with a reference to its parent layer $L_{k-1}$. The parent layer is in turn updated with the reference to the newly created layer. Then the layer is passed to the `initial_heuristic_layer` function of the configured `PDRHeuristic`. This function accepts a goal distance $k$ and a reference to the `Layer` instance and inserts clauses into the layer $L_k$ to strengthen it according to the implemented heuristic. The default `NoopPDRHeuristic` does not insert anything into the provided layer and returns immediately. This lack of seeding is used as a ground truth to compare against, since comparing the seeded performance characteristics to a fundamentally different implementation such as Sudas `PDRPlan`[4] would not be meaningful. The `PatternDBPDRHeuristic` uses a provided pattern database to seed clauses into the provided layer.

### 5.2.1 The Pattern Database PDR Heuristic

A pattern database, as implemented in Fast Downward, receives a pattern and seeds its heuristic values database using this pattern. Fast Downward implements various pattern generator algorithms, which use the internal task description to generate a suitable pattern according to some specified parameters. Through the modular nature of Fast Downward, the pattern generator can be chosen dynamically through a command line argument. Fast Downward version 22.06 implements the following pattern generators:

- CEGAR pattern generator
- greedy pattern generator
- manual pattern generator
- random pattern generator

Since the manual pattern generator is not strictly a generator but rather emits the manually specified pattern, we will omit it in our analysis. One of the parameters all pattern generator algorithms have in common, is the size of the resulting pattern database, as measured in the number of states in the projected state space. Since this number directly influences the number of clauses in the layers after seeding, this parameter must be chosen carefully.

A comparison of how the choice of pattern generators and their parameters affects the performance of the seeded PDR algorithm is shown in Chapter 6.

[4]PDRPlan is Sudas reference implementation of the PDR planning algorithm and can be found at `https://github.com/quickbeam123/PDRplan`

### 5.2.2 Seeding a Layer from the Pattern Database

Seeding a layer $L_k$ from the pattern database generally works as described in Section 4.3, however, some tweaks are necessary for compatibility with Fast Downward. The `PatternDatabase` class in Fast Downward does not offer an interface to access the states of the projected state space and their heuristic value directly. Instead, the method `get_value` is exposed, which expects a non-projected state and returns its heuristic value. Since the pattern $P$ is accessible, we can iterate over all projected states $s \in S_P$. Assigning the variables $v \in \mathcal{V} \setminus P$ of the projected state to a value of zero results in a valid non-projected state $s' \in S$. This is the case because the domain of a variable in Fast Downward always contains 0, since every domain has a size of at least two, and Fast Downward uses zero-based indexing for the domain values. If the queried value $h(s')$ is bigger than the specified goal-distance $k$, no state which has a projection equal to $s$ can ever have a goal distance of $k$ or less. The projected state $s$ is converted into a `LiteralSet` of type `SetType::CUBE` according to the rules of the *Lits* function, the positive literals are replaced by their negative equivalents as discussed in Section 4.3, and its inverse is added as a new positive clause to the layer $L_k$. After iterating over all projected states, all states that have a heuristic value of more than $k$ are not a model of $L_k$.

# 6 Evaluation

We have successfully implemented the Property-Directed Reachability algorithm as part of the Fast Downward planning system, as well as layer seeding based on the pattern database heuristic. In this chapter we will evaluate the performance of the seeded PDR algorithm. We compare the seeded algorithm using a pattern database that has been initialized with a variety of pattern generators to the non-seeded algorithm.

## 6.1 Evaluated Configurations

This section will list the various configurations that have been tested to evaluate the performance and behavior of the seeded PDR algorithm. A list of configurations and their corresponding configuration string, as well as the theoretical maximal number of projected states, can be found in Table 1.

**Non-Seeded PDR** This configuration uses the `NoopPDRHeuristic` as a seeding heuristic, and is used as the base for all comparisons. As mentioned in Section 5.2, this seeding heuristic does not modify the provided layer at all and therefore performs no seeding. We call this configuration the non-seeded PDR configuration or by its shorthand name `noop` from here on out.

**Greedy Pattern Generator** The configurations `greedy-50` to `greedy-1000` use the pattern database seeding heuristic as introduced in Section 5.2 with the greedy pattern generator. The parameter of the pattern generator refers to the maximum number of states the projected state space can contain. As listed in Table 1, the maximal number of projected states ranges from 50 to 1000. Usually, this is considered to be a very low number of states, yielding a very weak heuristic. However, since the number of projected states directly maps to the number of seeded clauses, and since the pattern size directly maps to the size of these clauses we need to limit this number to maintain performance. Ideally, we would increase the number of projected states to get better heuristic values and to restrict the layers more.

The greedy pattern generator selects as many variables as possible until the number of projected states is not within the predefined limit. The variables are selected in the following order: First, all variables occurring in the goal formula are selected, then variables are selected according to the order of the casual graph as introduced by Helmert [7].

**CEGAR Pattern Generator** The configuration `cegar` uses the pattern database seeding heuristic with the CEGAR [10] pattern generator. Similar to the greedy pattern generator, the CEGAR pattern generator exposes a configuration option to set the maximum number of projected states. However, for the planning problems used in the benchmarks, the CEGAR pattern generator never reaches more than 500 projected states, and therefore we left the setting at the default value of 1 million projected states.

**Random Pattern Generator** The configuration `rand` uses the seeding heuristic with the random pattern generator as described by Rovner et al. [10] in the

| Shorthand | Seed Heuristic | Max. Projected States |
|---|---|---|
| `noop` | `pdr-noop()` | - |
| `greedy-50` | `pdr-pdb(pattern=greedy(50))` | 50 |
| `greedy-100` | `pdr-pdb(pattern=greedy(100))` | 100 |
| `greedy-500` | `pdr-pdb(pattern=greedy(500))` | 500 |
| `greedy-1000` | `pdr-pdb(pattern=greedy(1000))` | 1000 |
| `cegar` | `pdr-pdb(pattern=cegar_pattern())` | 1'000'000 |
| `rand` | `pdr-pdb(pattern=random_pattern(1000))` | 1000 |

Table 1: List of seeding heuristic configurations used for testing.

experiment section. Just like the above pattern generators, the random pattern generator has a configuration option for the maximum number of projected states.

## 6.2 Benchmark Setup

In order to evaluate the performance of various configurations of the seeded PDR planner, we tasked them with solving a set of unit-cost problems from the "downward-benchmarks"[5] task collection. The full list of problem domains can be seen in the Appendix in Table 3 as part of the coverage table. Each benchmark involved running a PDR configuration with a planning task and collecting its run-time metrics, which were then used to compare different aspects of each planner configuration. The benchmarks were conducted on the SciCORE[6] scientific computing center at the University of Basel. Every node in the cluster has 6.4 GiB of RAM per core and runs on Intel Xeon Silver 4114 2.2 GHz processors.

We utilized the Downward Lab [11] Python library to set up the benchmarks, collect metrics from the produced log files, and help with the comparison of the collected metrics. Downward Lab offers tools to generate scatter plots that compare a property across multiple planner configurations pairwise, and offers interfaces to build custom visualizations.

Every configuration was tasked with solving every problem with the standard memory limit suggested by Downward Lab, and with a higher than standard time limit of two hours. The increased time limit was chosen to limit the number of tasks that are aborted due to the time constraint, so that we are able to compare problems where one configuration takes significantly longer than another configuration.

We use "relative" scatter plots to compare the value of a property at a run of a seeded PDR configuration versus the value of the same property on the same planning task using the non-seeded PDR configuration. The $x$-value of every point denotes the absolute value of the property for the non-seeded PDR configuration, whereas the $y$-value represents the relative value such that $x * y$ yields the value of the property for the compared PDR configuration. A value less than one indicates that the seeded PDR configuration exhibited a lower value for the specific property than the non-seeded PDR configuration.

[5] https://github.com/aibasel/downward-benchmarks
[6] https://scicore.unibas.ch/

| PDR Configuration | Successful | Out of Time | Failed |
|---|---|---|---|
| `noop` | 861 | 307 | 22 |
| `greedy-50` | 846 | 318 | 26 |
| `greedy-100` | 843 | 321 | 26 |
| `greedy-500` | 818 | 350 | 22 |
| `greedy-1000` | 799 | 372 | 19 |
| `cegar` | 865 | 303 | 22 |
| `rand` | 864 | 304 | 22 |

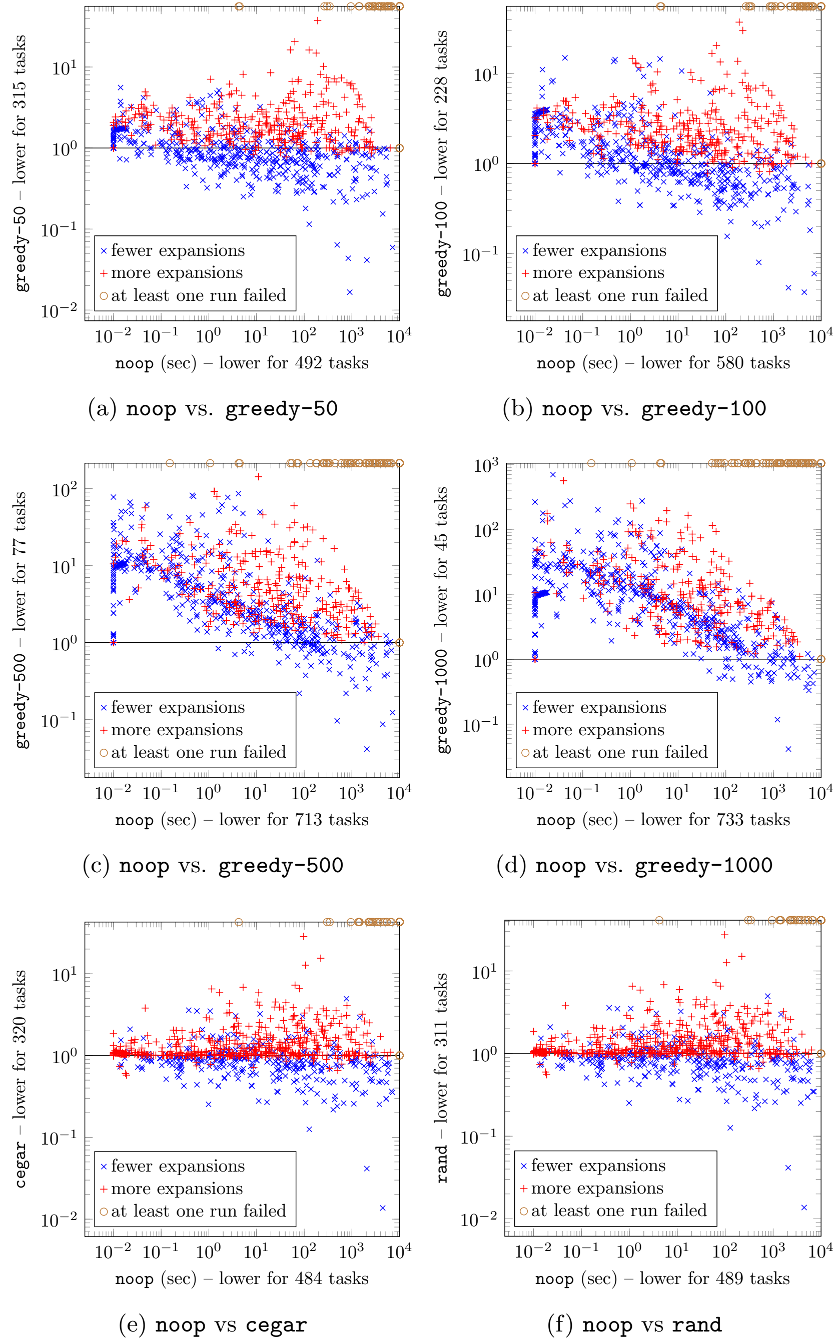
Table 2: The number of timed-out, successful, and failed planning tasks per PDR configuration. In total there are 1190 total tasks per configuration. Tasks are considered successful if a valid path is found or if it was shown that the task is unsolvable within the time and memory constraint. Failed tasks only consist of tasks that received the `sigkill` signal from the cluster. None of the tested planning tasks failed due to not having enough memory.

### 6.3 Results

The most important metric for the evaluation of the seeded PDR algorithm is how many of the tasks were able to be solved within the time and memory constraints. Table 2 shows that the `cegar` and `rand` configurations were able to solve the most tasks successfully. However, they only solved 4 and 3 additional tasks compared to the non-seeded `noop` configuration. This is surprising because we expected the seeded configurations to solve the tasks significantly faster than the base configuration. The worst configuration was the `greedy-1000` configuration, which, compared to the noop configuration, timed out on 65 more tasks. The failed runs shown in Table 2 are due to a `sigkill` signal that the planners received from the cluster. We were not able to fully determine the root cause of this, since this behavior was not reproducible on local machines. For all configurations, 13 of those failures happen when solving problems of the `organic-synthesis-opt18-strips` domain. Other affected domains were `satellite` with six to eight failed problems depending on the configuration, and `driverlog`, `logistics98`, `mystery` and `storage` with zero to two failed problems. We can not attribute those failures due to timeout conditions, since those errors happened within the defined time constraint. The total number of tasks that were successfully solved, grouped by domain, can be seen in the Appendix in Table 3.

To compare the performance in more detail, we measured the total time it takes for each planning task to be solved. Lower total planning times are better. We expected to find that seeding the layers results in decreased planning time in comparison to the non-seeded configuration. We also expect to see that "over-seeding" results in diminishing returns or even decreased performance due to increased calculation overhead.

As seen in Figure 5, the seeded PDR algorithm does not generally perform better than the non-seeded variant. While the `greedy-50`, `cegar`, and `rand` configurations perform on average similar to the non-seeded configuration, with a total planning time of 1.6 times, 1.3 times and 1.3 times more than the `noop` configuration, the variance of the search time increase is high with 4.3, 1.9 and




(a) noop vs. greedy-50 (b) noop vs. greedy-100

(c) noop vs. greedy-500 (d) noop vs. greedy-1000

(e) noop vs cegar (f) noop vs rand

Figure 5: Comparison of the total time between the non-seeded configuration and the seeded configurations using relative scatter plots. The blue marks signify that the task had fewer expanded obligations in the seeded configuration than in the non-seeded configuration. The red marks signify the opposite. The cluster of points at $x = 10^{-2}$ stems from rounding the total time up to a minimum of $10^{-2}$. The same comparison grouped by the problem domain can be found in the Appendix in Figure 11.

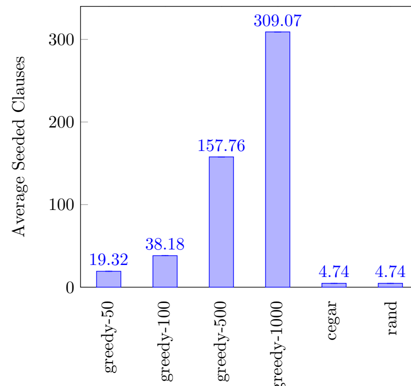



Figure 6: Number of seeded clauses for layer $L_0$, averaged over all tasks.

1.8 respectively. However, it is obvious that the configurations `greedy-100` to `greedy-1000` do worse than the non-seeded configuration for most tasks, with an average planning time increase of more than five times. This slowdown for the `greedy-1000` configuration is especially pronounced for smaller tasks, where the total planning time for most tasks is over twenty times higher. While our hypothesis of seeding leading to faster solve times was not confirmed, we can see that over-seeding due to too many inserted clauses does indeed occur. The slowdown of the seeded configurations generally matches the number of total seeded clauses as seen in Figure 6. However, a general trend for configurations `greedy-500` and `greedy-1000` can be seen: the longer the non-seeded configuration takes to solve a task, the closer the planning time approaches the non-seeded planning time. This suggests that for long-running tasks, the layer sizes are increasing enough over time so that the added clauses from the seeding step have less of a negative relative performance impact.

We did not find any significant correlation between certain domains and any increase or decrease in planning performance. This can be seen in the Appendix in Figure 11.

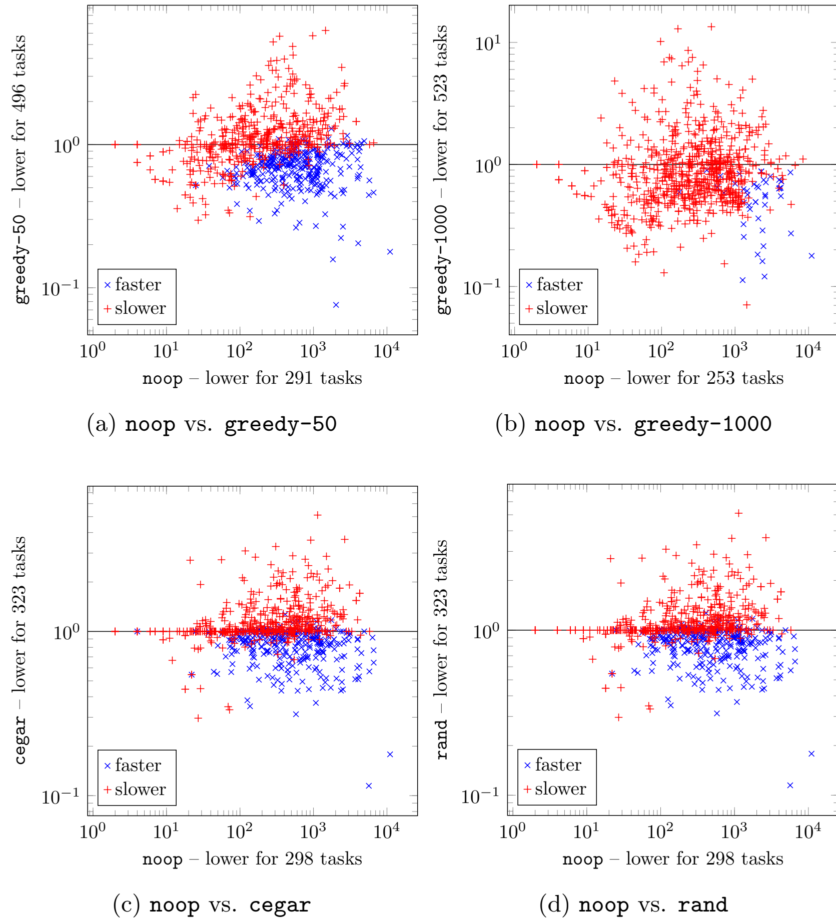
Another metric is the number of obligation expansions for each configuration. We expected seeding to lead to strictly fewer expanded obligations due to the layers being more strict from the beginning of the planning phase. As seen in Figure 7 this is however not the case. While for all seeded configurations the majority of tasks have fewer expanded obligations than the non-seeded configuration, there are still a significant number of tasks where more obligations are expanded in the seeded configuration. This raises the question of how stricter layers can lead to more expansions.

A possible explanation could be that due to the layers being strict from the beginning, the PDR algorithm does not enter the path construction phase until later iterations. The path construction phase is only entered in an iteration $k$ if $s_I \models L_k$. Since we already seeded layer $L_k$ this is only the case for iterations where $h_{pdb}(s_I) \leq k$. This could prevent the algorithm from finding some high-quality reasons $R$ through the *extend* function. Missing the clauses $\neg r \in R$ from the layer would in turn lead to more expansions in later iterations.

Those tasks that were solved faster in the seeded configurations almost exclusively consist of tasks that expanded fewer obligations in the seeded config-

urations as seen in Figure 7. This demonstrates the link between the planning time and the number of expanded obligations. While faster tasks mostly consist of tasks that also expanded fewer obligations, the opposite is not generally true. In the configurations `greedy-50` to `greedy-1000`, many of the tasks that expanded fewer obligations are still slower than in the `noop` configuration. Interestingly, this is much less the case in the `cegar` and `rand` configurations. Tasks with fewer expanded obligations are, except for some individual cases, faster in those configurations.

Figure 8 compares the size of the initial layer $|L_0|$ in multiple configurations. The size of the first layer is also the number of unique clauses in all layers $|L_0| = |\cup_{i>0} L_i|$ which follows from layer invariant 2. There is a direct correlation between the number of clauses and the planning time. Over 71% of tasks that have fewer clauses in the seeded variant are also faster in that variant, and inversely over 86% of tasks that have more clauses are also slower. This finding is consistent with the findings in Sudas paper that bigger layers lead to worse performance and is the reason why Suda goes to great length to reduce the size of clauses through subsumption and reason minimization.



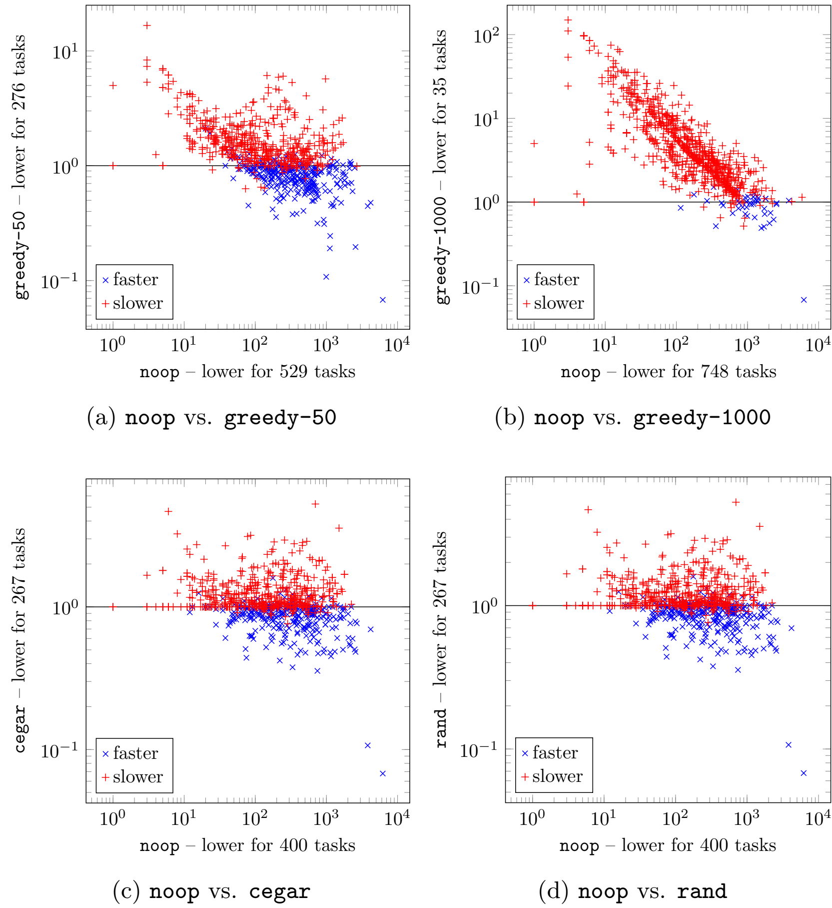

(a) `noop` vs. `greedy-50`

(b) `noop` vs. `greedy-1000`

(c) `noop` vs. `cegar`

(d) `noop` vs. `rand`

Figure 7: Comparison of the number of expanded obligations for the non-seeded and the seeded PDR configurations using relative scatter plots. The blue marks signify that the task was solved faster in the seeded configuration than in the non-seeded configuration. The red marks signify the opposite. The plots for the configurations `greedy-100` and `greedy-500` are excluded from from this figure for the sake of clarity. They show an interpolation of the data between `greedy-50` and `greedy-1000` and can be found in the Appendix as Figure 9. Tasks where at least one of the configurations failed are not shown.




(a) `noop` vs. `greedy-50`

(b) `noop` vs. `greedy-1000`

(c) `noop` vs. `cegar`

(d) `noop` vs. `rand`

Figure 8: Comparison of the size of the first layer $|L_0|$ after planning has concluded using relative scatter plots. The blue marks signify that the task was solved faster in the seeded configuration than in the non-seeded configuration. The red marks signify the opposite. The plots for the configurations `greedy-100` and `greedy-500` are excluded from from this figure for the sake of clarity. They show an interpolation of the data between `greedy-50` and `greedy-1000` and can be found in the Appendix as Figure 10. Tasks where at least one of the configurations failed are not shown.

# 7 Conclusion

The goal of this thesis was to implement layer seeding using heuristics as an additional step of the PDR algorithm for planning and to test our hypothesis that seeding improves performance by reducing the number of obligation expansions. We implemented the SAT-free PDR variant for planning proposed by Suda as a search engine in the Fast Downward planning system and added heuristics-based seeding using the pattern database heuristic. We measured the performance characteristics of the non-seeded implementation and compared it to a variety of seeded configurations. Those configurations varied by the choice of pattern generators and number of projected states.

We found that seeding generally does neither consistently improve the total planning time, nor does it consistently reduce the number of expanded obligations. Additionally, we found that heavily seeding the layers leads to a significant increase in planning time for small tasks, while for big tasks we found the time increase to be less severe. For some bigger planning tasks, the heavily seeded configuration leads to the same planning time as the non-seeded configuration, for some occasional tasks even a significant time decrease. We also found that the lightly seeded configurations perform on average slightly slower than the non-seeded configuration. However, while many of the tasks have a significantly higher planning time, many others have an equally significant reduction in planning time. In terms of the number of expanded obligations, we found that a significant number of tasks have an increased number of obligation expansions in the seeded configuration. We hypothesize that this is due to high-quality reasons that were never returned from the *expand* function, due to the path construction phase not being executed for low-index layers in the seeded configuration.

While our approach to seeding did not yield the desired results, there are several areas of optimization that could be investigated in future research: For example, it may be possible to further reduce the size and number of seeded clauses by implementing clever clause minimization strategies. Another promising approach would be to combine multiple small pattern databases, in order to find seed clauses consisting of only a small number of literals. Additionally, implementing seeding using other heuristics, such as the merge and shrink heuristic [3][12], may also prove to be worthwhile.

# References

[1] Aaron R. Bradley. “SAT-Based Model Checking without Unrolling”. In: *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation.* 2011, pp. 70–87.

[2] Joseph C. Culberson and Jonathan Schaeffer. “Pattern Databases”. In: *Computational Intelligence* 14.3 1998, pp. 318–334. DOI: 10.1111/0824-7935.00065.

[3] Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. “Directed Model Checking with Distance-Preserving Abstractions”. In: *International Journal on Software Tools for Technology Transfer* 11.1 2009, pp. 27–37. DOI: 10.1007/s10009-008-0092-z.

[4] Niklas Eén, Alan Mishchenko, and Robert Brayton. “Efficient Implementation of Property Directed Reachability”. In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design.* 2011, pp. 125–134.

[5] Richard Fikes and Nils J. Nilsson. “STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving”. In: *Artificial Intelligence* 2.3 1971, pp. 189–208. DOI: 10.1016/0004-3702(71)90010-5.

[6] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. “Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning”. In: *AAAI* 2007, pp. 1007–1012.

[7] Malte Helmert. “A Planning Heuristic Based on Causal Graph Analysis”. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 16 2004, pp. 161–170.

[8] Malte Helmert. “Concise Finite-Domain Representations for PDDL Planning Tasks”. In: *Artificial Intelligence* 173.5 2009, pp. 503–535. DOI: 10.1016/j.artint.2008.10.013.

[9] Malte Helmert. “The Fast Downward Planning System”. In: *Journal of Artificial Intelligence Research* 26 2006, pp. 191–246. DOI: 10.1613/jair.1705.

[10] Alexander Rovner, Silvan Sievers, and Malte Helmert. “Counterexample-Guided Abstraction Refinement for Pattern Selection in Optimal Classical Planning”. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 29 2019, pp. 362–367. DOI: 10.1609/icaps.v29i1.3499.

[11] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. *Downward Lab.* 2017. DOI: 10.5281/zenodo.790461.

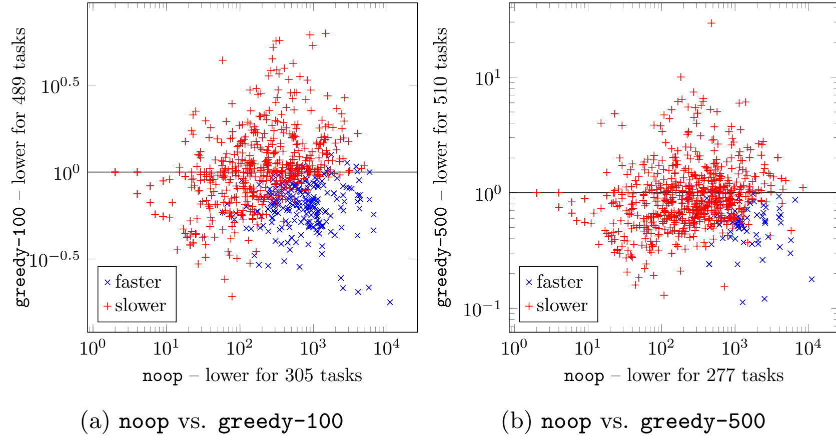
[12] Silvan Sievers and Malte Helmert. “Merge-and-Shrink: A Compositional Theory of Transformations of Factored Transition Systems”. In: *Journal of Artificial Intelligence Research* 71 2021, pp. 781–883. DOI: 10.1613/jair.1.12557.

[13] Martin Suda. “Property Directed Reachability for Automated Planning”. In: *Journal of Artificial Intelligence Research* 50 2014, pp. 265–319. DOI: 10.1613/jair.4231.

# A Appendix

| | noop | greedy-50 | greedy-100 | greedy-500 | greedy-1000 | cegar | rand |
|---|---|---|---|---|---|---|---|
| airport (50) | 21 | 22 | 22 | 21 | 22 | 22 | 22 |
| barman-opt14-strips (14) | 14 | 14 | 14 | 14 | 12 | 14 | 14 |
| blocks (35) | 35 | 35 | 35 | 34 | 34 | 35 | 35 |
| childsnack-opt14-strips (20) | 5 | 1 | 2 | 0 | 0 | 2 | 2 |
| depot (22) | 18 | 18 | 18 | 18 | 18 | 18 | 18 |
| driverlog (20) | 15 | 15 | 15 | 15 | 15 | 15 | 15 |
| freecell (80) | 51 | 40 | 40 | 29 | 25 | 49 | 49 |
| grid (5) | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| gripper (20) | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| hiking-opt14-strips (20) | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| logistics00 (28) | 28 | 28 | 28 | 28 | 28 | 28 | 28 |
| logistics98 (35) | 23 | 23 | 24 | 23 | 23 | 23 | 23 |
| miconic (150) | 150 | 150 | 150 | 150 | 150 | 150 | 150 |
| movie (30) | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| mprime (35) | 32 | 30 | 30 | 30 | 30 | 33 | 33 |
| mystery (30) | 25 | 25 | 24 | 20 | 18 | 23 | 23 |
| nomystery-opt11-strips (20) | 10 | 10 | 8 | 8 | 10 | 9 | 9 |
| openstacks-strips (30) | 26 | 26 | 26 | 26 | 25 | 26 | 26 |
| organic-synthesis-opt18-strips (20) | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| parking-opt11-strips (20) | 2 | 1 | 1 | 3 | 3 | 2 | 2 |
| parking-opt14-strips (20) | 0 | 2 | 2 | 0 | 0 | 0 | 0 |
| pathways (30) | 29 | 29 | 29 | 27 | 27 | 30 | 30 |
| pipesworld-notankage (50) | 37 | 37 | 35 | 34 | 33 | 40 | 39 |
| pipesworld-tankage (50) | 24 | 25 | 25 | 21 | 20 | 24 | 24 |
| psr-small (50) | 50 | 50 | 50 | 50 | 50 | 50 | 50 |
| rovers (40) | 32 | 32 | 32 | 31 | 31 | 31 | 31 |
| satellite (36) | 19 | 19 | 17 | 19 | 18 | 22 | 22 |
| snake-opt18-strips (20) | 11 | 10 | 8 | 10 | 8 | 10 | 10 |
| storage (30) | 22 | 21 | 23 | 22 | 22 | 22 | 22 |
| termes-opt18-strips (20) | 5 | 4 | 4 | 5 | 5 | 6 | 6 |
| tidybot-opt11-strips (20) | 13 | 14 | 14 | 13 | 12 | 13 | 13 |
| tidybot-opt14-strips (20) | 6 | 7 | 7 | 5 | 2 | 6 | 6 |
| tpp (30) | 17 | 17 | 17 | 18 | 18 | 18 | 18 |
| trucks-strips (30) | 10 | 10 | 11 | 13 | 9 | 11 | 11 |
| visitall-opt11-strips (20) | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| visitall-opt14-strips (20) | 13 | 14 | 15 | 14 | 14 | 15 | 15 |
| zenotravel (20) | 18 | 17 | 17 | 17 | 17 | 18 | 18 |
| **Total (1190)** | 861 | 846 | 843 | 818 | 799 | 865 | 864 |

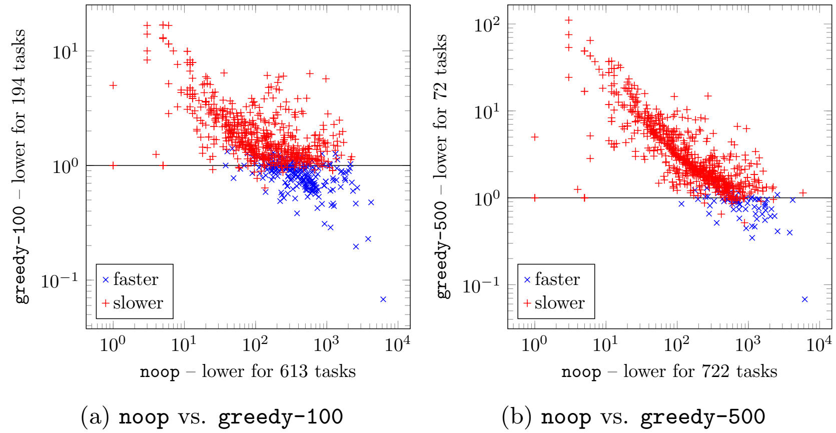
Table 3: Coverage of all tested domains and algorithm configurations. Tasks are considered successful if they find a valid path or show that the task is unsolvable before the maximal time is reached.




(a) `noop` vs. `greedy-100`

(b) `noop` vs. `greedy-500`

Figure 9: Comparison of the number of expanded obligations for the non-seeded and the seeded PDR configurations. The blue marks signify that the task was solved faster in the seeded configuration than in the non-seeded configuration. The red marks signify the opposite.

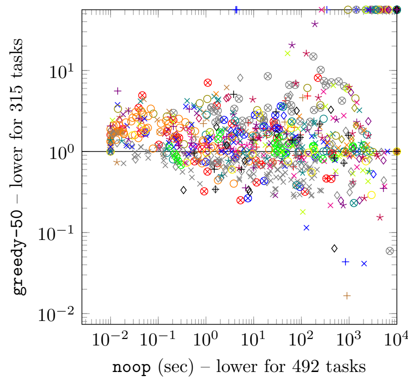



(a) `noop` vs. `greedy-100`

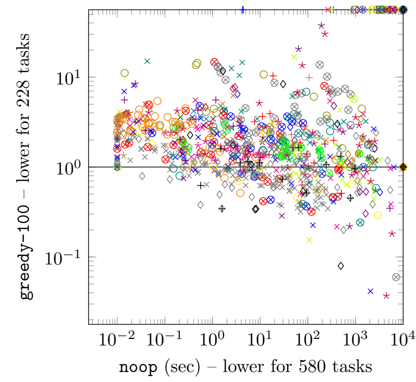
(b) `noop` vs. `greedy-500`

Figure 10: Comparison of the size of the first layer $|L_0|$ after planning has concluded. The blue marks signify that the task was solved faster in the seeded configuration than in the non-seeded configuration. The red marks signify the opposite.

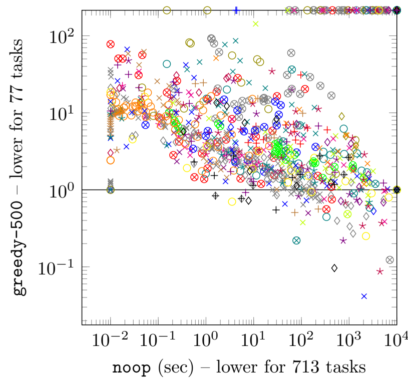
× airport, + barman-opt14-strips, ⊗ blocks, + childsnack-opt14-strips, ⋆ depot, × driverlog, ⊗ freecell, ◇ grid, ⋆ gripper, ⊗ hiking-opt14-strips, × logistics00, ○ logistics98, × miconic, ⋆ movie, × mprime, ○ mystery, + nomystery-opt11-strips, ⊗ openstacks-strips, ◇ organic-synthesis-opt18-strips, ⊗ parking-opt11-strips, + parking-opt14-strips, × pathways, ⋆ pipesworld-notankage, × pipesworld-tankage, ○ psr-small, ◇ rovers, ○ satellite, ◇ snake-opt18-strips, × storage, ⊗ termes-opt18-strips, × tidybot-opt11-strips, ◇ tidybot-opt14-strips, ⋆ tpp, ◇ trucks-strips, ◇ visitall-opt11-strips, + visitall-opt14-strips, + zenotravel



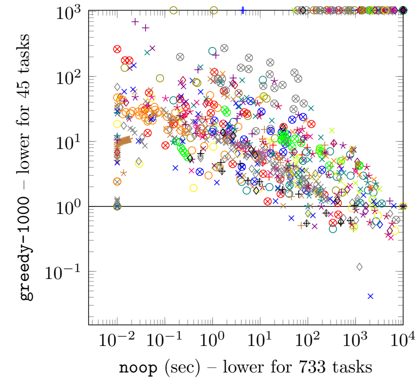

(a) `noop` vs. `greedy-50`



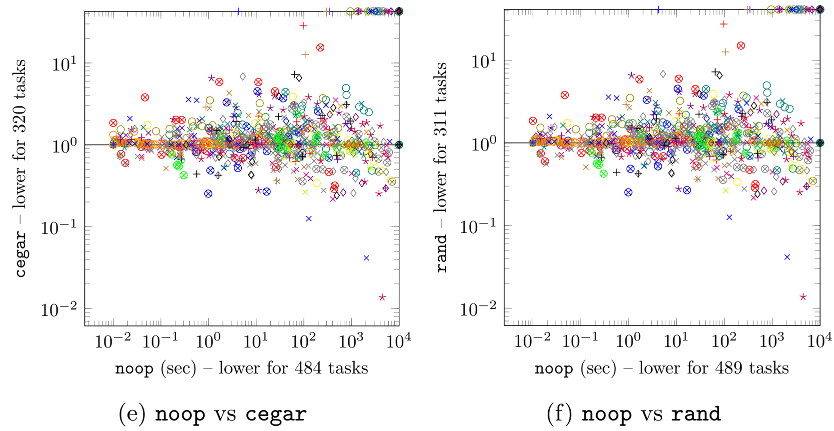

(b) `noop` vs. `greedy-100`




(c) `noop` vs. `greedy-500`




(d) `noop` vs. `greedy-1000`




(e) `noop` vs `cegar` (f) `noop` vs `rand`

Figure 11: Comparison of the total time between the non-seeded configuration and the seeded configurations using relative scatter plots. The markers are grouped by the problem domain.