Advancing Property Directed Reachability for Classical and Fully Observable Nondeterministic Planning

Ava Clifton

A thesis submitted for the degree of

Doctor of Philosophy

The Australian National University

College of Engineering, Computing and Cybernetics.

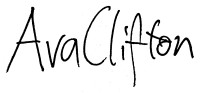
26 October 2025

To family, friends, and the wonderful universe we are intimately connected with

Disclaimer

This dissertation is an account of research undertaken between March 2020 and January 2025 at the Research School of Computing, The Australian National University, Canberra, Australia.

This thesis is based in part on papers published in refereed scientific workshops or conferences, or under review at such a conference. The development of ideas and research was undertaken with guidance from my supervisors Charles Gretton, Patrik Haslum and Alban Grastien, as well as Jussi Rintanen, and published papers that describe research in this thesis were written collaboratively with them.



Ava Clifton

26 October 2025

Acknowledgements

This work has benefited greatly from the influence of many people.

First and foremost, this work would not have been possible without the support of my supervisor Charles Gretton, whose interest, expertise and energy is inspiring. I want to thank him for the countless hours he has given to me over my studies.

I would like to thank the rest of my supervisory panel, Patrik Haslum and Alban Grastien, for their very valuable input over these years. Additionally, I would like to thank Jussi Rintanen for inviting me to visit him in Helsinki, and his help thereafter.

This research was undertaken with the assistance of resources from the National Computational Infrastructure (NCI Australia), an NCRIS enabled capability supported by the Australian Government.

I would like to thank Lia Langman for all her support over the years, as well as the many other friends who have helped me along the way.

Most importantly I would like to thank my family for their lifelong support, especially my parents Jessica and Stephen Clifton.

Abstract

Property Directed Reachability (PDR) is a powerful search paradigm that has been developed in the 21st century for model checking hardware and software systems, as well as for automated planning. To solve classical planning problems, PDR searches iteratively solve a series of small decision problems about whether a particular state can be progressed a single step toward the goal. Unlike earlier monolithic bounded model checking paradigms, which form the basis of many existing SAT-based planning tools, PDR does not unroll the system transition function over multiple steps, reasoning only about one step at a time. This approach results in a smaller memory footprint compared to monolithic bounded model checking. Additionally, PDR can detect when a problem has no solution without requiring a completeness threshold. However, by not reasoning about multi-step formulae, PDR misses out on the powerful multi-step propagation that underpins the strong performance of some SAT-based planning procedures. To address this, in our first contribution we develop two new PDR algorithms for classical planning that operate by solving small multi-step decision problems rather than single-step problems. Our results show that in certain domains a multi-step approach leads to more efficient search.

We then turn to parallel computing environments to further explore how to accelerate PDR. We introduce new parallel PDR algorithms for classical planning that significantly outperform serial procedures and existing parallel baselines from the model-checking literature. Our first algorithm, called Parallel State PDR (PS-PDR), evaluates many decision problems in parallel using a pool of incremental SAT workers. PS-PDR distinguishes itself from other distributed PDR algorithms by centrally maintaining a single queue of decision problems, enabling a more focused and efficient search compared to a PDR portfolio approach. Our second algorithm, Parallel Decomposition PDR (PD-PDR), takes a compositional approach, decomposing the problem into subproblems and sequencing them according to the concrete problem structure. These subproblems are solved in parallel, and a concrete plan is constructed by combining the individual subproblem solutions. In problems with compositional structure, this can result in orders of magnitude speedup.

Finally, we extend PDR for the first time to tackle the Fully Observable Non-Deterministic (FOND) planning problem, which generalizes classical planning by allowing actions to have multiple possible outcomes chosen nondeterministically at execution. We evaluate our algorithm against leading FOND planners across a range of benchmarks, including cases where strong cyclic policies do and do not exist. Our results demonstrate the competitiveness of our approach, and the effectiveness of using PDR in planning under uncertainty.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Automated planning and related fields study what dynamical systems can do, how to achieve a desirable system state, and algorithms for efficiently answering such questions about dynamical systems. They equally study what systems cannot do, sometimes proving that certain states cannot be reached from other states. Automated planning systems are widely used, including for analysing various complicated logistical operations. They are also often used in critical systems and in particular are used to prove that these systems will work in certain ways, or otherwise show how these systems can fail.

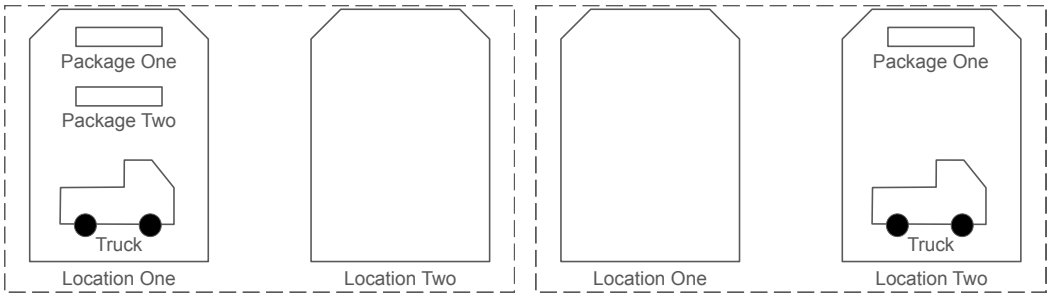
Much of this thesis limits itself to *classical* planning; that is systems with:

- A finite set of *propositions*, binary variables representing aspects of the system.
- A finite set of *states*, each being a set of assignments to the propositions.
- A finite set of *actions* each of which transitions the system between states.
- An *initial* state.
- A *goal condition*.

Classical planning is then the task of determining if there exists a sequence of actions which can transition the initial state to a state which satisfies the goal condition. If such a sequence of actions exists, it may be useful to also produce this sequence, or find a sequence which minimizes some cost function – e.g. the shortest such sequence. In the case that no such sequence exists, it may be useful to produce a proof of this. We present a classical planning problem in Example 1 below.

Example 1 (A Logistic Problem)
Consider a problem where there are two locations, two packages, and one truck. Packages and the truck can be at either location, additionally the packages can be in the truck. The truck has infinite capacity. The available actions are to drive the truck from any location to any other location, and to load/unload packages to/from the truck, at a truck's current location.

Initially, the truck and both packages are at location one, and our goal condition is that the truck and package one are both at location two, shown graphically in Figure 1.1. In order to achieve this goal, a valid plan would be to load package one into the truck, drive the truck with package one to location two, then unload the package there.



The initial state. The goal condition.

Figure 1.1: Logistics example initial state and goal condition.

We also consider *Fully Observable Non-Deterministic* (FOND) Planning. FOND planning is an extension of classical planning used to model nondeterminism within the system. FOND planning behaves similarly to classical planning with two key differences. First, when an action is executed, instead of a sole successor state being produced, one of many *outcomes* could be encountered, chosen by the environment. FOND can be used to model, for example, a procedure which may or may not be successful, and have a success and fail state as outcomes. Secondly, what counts as a solution in FOND planning differs from classical planning. There are three main classes of solution:

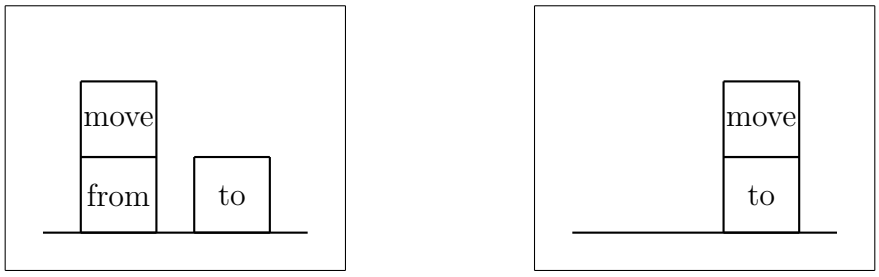
- *Weak solutions* are those where it is possible, but not guaranteed to be able to reach a goal state. Whether a goal state is reached is dependant on which outcomes are chosen by the environment. For example if trying to move between two close locations, a weak plan could be to drive, if the weather permits it. Whether this plan is successful or not is dependant on the weather.
- *Strong solutions* are those where it is guaranteed that a goal state will be reached, and all contingencies are planned for. Additionally, the number of actions taken to transition from the initial state to the goal state will be below some finite bound.
- *Strong cyclic solutions* are those where it is guaranteed that a goal state will be reached eventually. This is dependant on the *fairness assumption*, that if a nondeterministic action is taken repeatedly on a state, the number of applications needed to end up at a given outcome will always be finite. I.e.

if an action is repeatedly applied to a state, each outcome will eventually be encountered, and after that encounter, will eventually be encountered again, and so forth. For example, when trying to move between two close locations, as above, if every day an attempt is made to drive if the weather permits, then if the weather abides by the fairness assumption then eventually the weather will be suitable and the plan will be successful. However, this can lead to arbitrarily long sequences of actions to get to a goal state.

This type of solution is the type we consider in this thesis. In Chapter 6 we develop a new solver to find strong cyclic solutions for FOND planning problems, or to otherwise prove none exist.

We present a FOND planning problem in Example 2.

Example 2 (A FOND Blocksworld Problem)
Consider a problem where there is a table and three blocks. Each block can either be on the table or on another block. We can use a robot arm to pick up blocks from the table or from another block, hold them, then put them down on the table or another block. We call the three blocks from, to and move. Initially, to and from are both on the table, and move is on from. Our goal condition is then to have move on to. We show this graphically in Figure 1.2.



Initial State Goal Condition

Figure 1.2: Blocksworld example initial state and goal condition.

However, the robot arm is clumsy. Whenever it goes to pick up or put down a block it may instead drop it on the table. As such, finding a strong cyclic solution involves accounting for the cases where the block is dropped. A strong cyclic policy for this problem is shown graphically in Figure 1.3. Note that when all blocks are on the table, and an attempt is made to pick up move, then if it fails the resulting outcome is to have all blocks on the table again. Achieving this outcome is equivalent to doing no action. If the action is applied repeatedly, then eventually the block will be successfully picked up, so in effect we think of this action being deterministic.

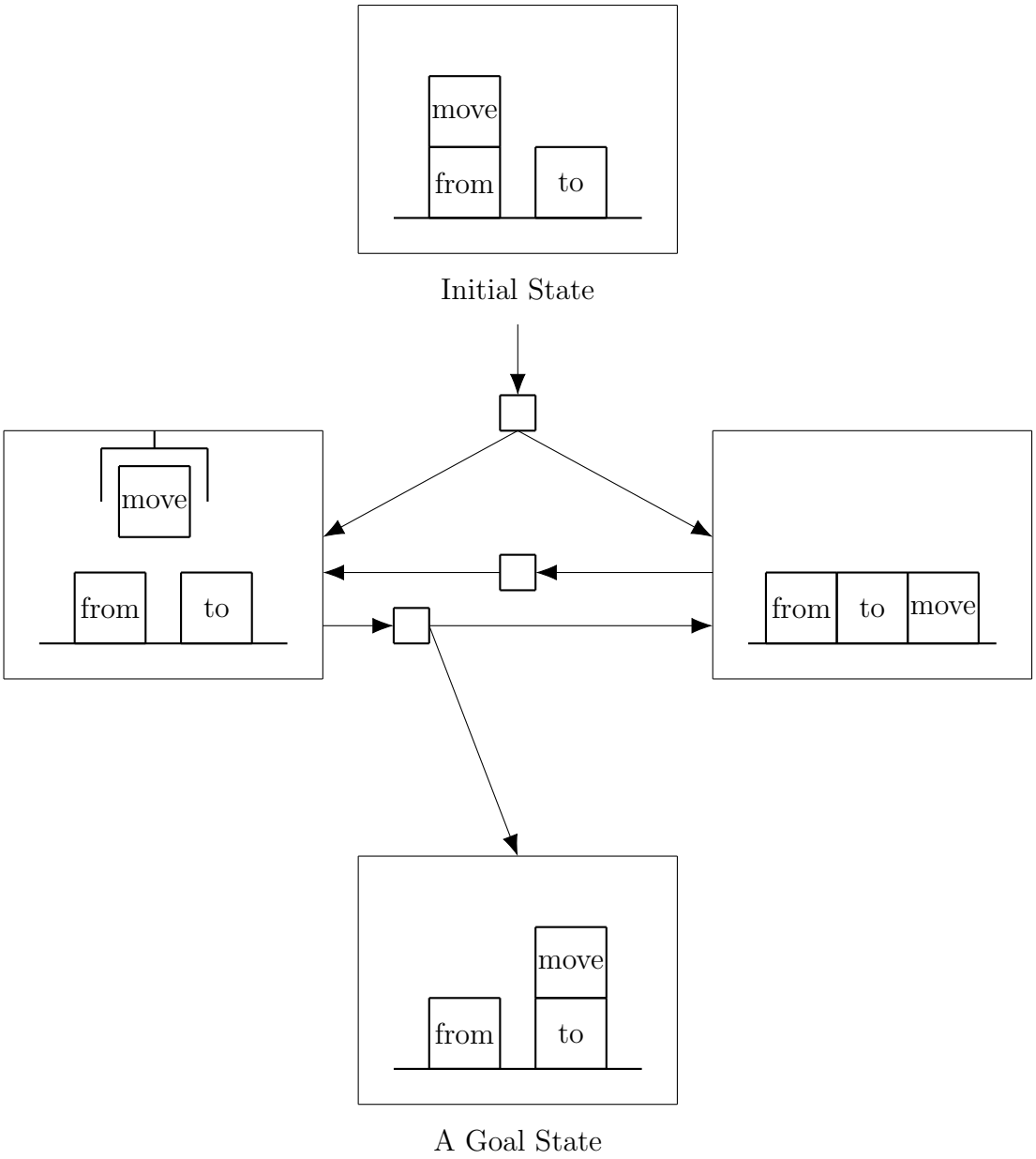


Figure 1.3: Blocksworld example strong cyclic policy. Actions are shown as outgoing arcs from states to a square. Outcomes of actions are shown as outgoing arcs from these squares.

1.1 Planning Approaches Overview

1.1.1 State Based Planning

State based planning (Ghallab et al. 2004a; Russell and Norvig 2016) procedures first consider the initial state, and search over actions that are applicable to that state. They then apply such actions to create a set of successor states to be explored. Such exploration continues for discovered states until a state satisfying the goal condition is found. Often an A* or other heuristic driven search is used and much research

attention has been put into finding suitable heuristics. We do not consider state based planning in the sequel.

1.1.2 SAT Planning

Boolean Satisfiability (SAT) is the NP-Complete problem of determining if there exists an interpretation of a set of Boolean variables which satisfies a given Boolean formula. Much research has gone into creating efficient SAT solvers, and while the SAT problem is NP-Complete, instances of the SAT problem are often efficiently solvable in practice. *SAT planning* consists of converting a planning problem into a SAT problem, and solving that SAT problem directly. Planning is PSPACE complete (Bylander 1994) and there is no known mapping from a planning problem onto a SAT problem. Instead, bounds are placed on the planning problem, and these bounded versions are solved via SAT. There are many strategies for which bounds to use, and when. Additionally, with certain bounds it is possible create a bounded planning problem at a *completeness threshold* that is solvable iff the original planning problem is solvable. This can be used to prove that the problem is unsolvable.

1.1.3 Property Directed Reachability (PDR)

Property Directed Reachability (PDR) is a planning procedure which maintains a compact representation of what states can reach the goal in a single step. Furthermore it also maintains representations of what states can reach the goal in n steps. This information is iteratively refined until a plan is found, or it can show that the initial state cannot reach the goal with any number of steps. This information is used to guide the search.

In the course of its execution PDR considers a set of states along with an optimistic estimate of how many steps each would need to transition to a goal state, known as a *queue*. PDR then repeatedly queries if any state on the queue can be transitioned to a state one step closer to the goal. This query is represented as a SAT problem, and answered using many of the techniques offered by SAT planning. If such a transition exists, then the successor state is added to the queue. If no such transition is possible, then this information is stored and used to constrain future queries. This algorithm, and its proposed modifications is the main focus of this thesis.

1.2 Thesis Outline and Contributions

The main contribution of this thesis is the development, implementation and evaluation of a collection of solvers all based on PDR. We find PDR to be a particularly interesting and useful algorithm to further develop here. It is sound and complete, and able to prove that no plan exists without the need to unroll the transition function, or have a completeness threshold. SAT solving for planning has been shown to be very effective, and PDR uses these techniques extensively while providing additional structure to help guide the search. Existing PDR implementations have been shown to be effective in hardware/software model checking. PDR has also been noted as being particularly amenable to parallelism. For these reasons we find PDR a particularly interesting algorithm to develop, and do so in the following chapters.

Chapter 3: Multi-Step SAT in PDR

We introduce two sound and complete solvers based on PDR, *PDR Macro* (PDR-M) and *PDR Interleaved Layers* (PDR-IL). In classical PDR, each SAT call corresponds to a transition from a known state to a state satisfying some successor constraints. Each of these SAT calls are completed almost instantly and many such calls are made in the course of the execution of PDR. SAT however is known to be a powerful tool when solving multi-step planning problems, so perhaps their prowess is being underutilized solving a single step planning problem. The motivation for this chapter is to consider the benefits that might be obtained if the SAT calls are larger, offloading some of the processing from the PDR process to the highly effective SAT solvers. If taken to its extreme, this could effectively remove the PDR logic from the algorithm, reducing it back to a traditional SAT planning approach. We suppose there is a middle ground which can still utilize PDR, while also benefiting from the SAT solver's ability to solve large problems effectively. We here offload some of the processing traditionally done by PDR to the SAT calls by allowing multiple transitions.

In the operation of PDR, the successor state being determined must satisfy some given successor constraints. PDR-M operates by having multiple transition steps where the first state is the given state being progressed, and the final state must satisfy the given successor constraints. PDR-IL operates differently, constraining each successive state to be consistent with a corresponding set of successor constraints being maintained by PDR. The performance of these approaches is highly variable, and dependant on the domain and the number of steps taken. With these solvers

we show a significant speedup in some instances and domains.

Chapter 4: Parallelizing Obligation Processing in PDR

We introduce a sound and complete solver *Parallel State PDR* (PS-PDR). Classical PDR iteratively takes a single state from the queue of work to be done, and processes it before continuing. PS-PDR processes many states independently in parallel with a collection of worker nodes. There exists parallel implementations of PDR, but crucially, these all:

- Are not made for planning problems.
- Employ a portfolio approach, having a collection of PDR processes running independantly sharing information, as opposed to a centralized queue as we do.
- Do not employ *obligation rescheduling* (introduced in 2.5.2), a technique explained later which is very important for the efficiency of PDR.

Because these existing parallel solvers are not made for planning, we reimplement them in order to evaluate the effectiveness of our new algorithm. We find PS-PDR to offer runtime performance improvements over serial PDR, as well as our implemented baseline portfolio solver.

Chapter 5: Parallel Decompositional PDR

We introduce a sound and complete solver *Parallel Decompositional PDR* (PD-PDR), which operates by decomposing the planning problem into a sequence of subproblems, so that a concrete problem solution corresponds to a concatenation of subproblem plans. Candidate subproblems are initially identified and sequenced according to a partition of problem variables, indicated by dependency analysis. Each subproblem is solved independently in parallel using a PDR algorithm. If the partition does not yield a concrete plan, either the problem is proved unsolvable as indicated by the unsolvability of a subproblem, or the process repeats on a contracted partition until a plan is produced or the concrete planning problem is proved unsolvable. Where problems are susceptible to decomposition, we find PS-PDR to provide significant runtime reductions, often even greater than those offered by PS-PDR.

Chapter 6: PDR for FOND

We introduce a sound and complete solver for FOND problems based on PDR that we call FOND-PDR. Many parts of the operation of PDR need to be adapted to work in the nondeterminism of a FOND setting. We can describe the key contributions of FOND-PDR over traditional PDR as:

- Modifying what it means for states to be progressed, and what is done with the result.
- Modifying the order in which states are progressed from the queue, to account for the potentially cyclical nature of FOND planning.
- Modifying what it means to have found a solution, in a FOND setting.

We find FOND-PDR can outperform other leading FOND planners in some domains, with far superior performance compared to incumbent FOND planners in a synthesised Blocksworld domain.

Appendix A

This appendix presents a table of all solvers introduced in this thesis, their introduction location in the text, and where they can be found.

Chapter 2

Background and Related Work

In this chapter we review formalisms and algorithms referred to in the rest of this thesis. The advancements made in this thesis that are dependant on these concepts are introduced in their associated chapters.

2.1 Boolean (SAT)isfiability

Boolean Satisfiability, often referred to as just *SAT*, is the problem of determining if there exists an interpretation of propositions which satisfies a given Boolean formula (Biere et al. 2009). The problem is NP-Complete, and was the first problem to be proven so (Cook 1971). There have been significant improvements in SAT solving and current solvers are some of the best for solving bounded decision problems (Biere et al. 2009). Because of this, many problems in planning and model checking are solved by first converting a bounded version of the problem to SAT (Zarpas 2005). SAT is used as the underlying mechanism in all solvers presented in this thesis.

2.1.1 Notation

For a proposition p we write $\neg p$ to denote the negation of p. A *literal* is either a proposition or its negation. For two formulae F and F', we write $F \wedge F'$ to denote the conjunction (and) between F and F', and likewise write $F \vee F'$ to denote the disjunction (or). We write $\Sigma(F)$ to denote the set of all propositions mentioned in a formula F. We write $a \Rightarrow b$ to denote that *a implies b*.

For simplicity and consistency, Boolean formulae are often represented in a normal form. *Conjunctive Normal Form* (CNF) is a simple form for which computing SAT is NP-Complete, and corresponds to the input format of most modern SAT solvers (Biere et al. 2009). CNF has the Boolean formula represented as a conjunction of disjunct literals. In CNF form, a disjunction of literals is known as a *clause*. All clauses have to be satisfied by setting at least one of their literals to *True*. An

example formula in CNF is demonstrated in Example 3.

Example 3 (A Boolean Formula in CNF)
Where, a, b, c, d and e are propositional variables, a valid formula in CNF would be:

$$(a \vee c \vee \neg d) \wedge (a \vee \neg b) \wedge (c \vee \neg d \vee \neg e)$$

A valid assignment would then be:

$$\begin{aligned} a &= True \\ b &= False \\ c &= True \\ d &= False \\ e &= False \end{aligned}$$

The removal of clauses from the problem instance will never decrease the solution space, just as adding a clause will never increase the solution space. The *empty clause* is the clause containing no literals, and is, as such, impossible to satisfy, so any formula containing this clause is impossible to satisfy. A *unit clause* is a clause containing a single literal. In order for a SAT instance to be satisfied, the variable in each unit clause must be assigned so that the clause is satisfied – i.e. if the literal is positive in the clause then the variable must be assigned *True*, and *False* otherwise. Any Boolean formula can be represented in CNF (Howson 2005).

For a formula F and assignment v, we write `SAT(F)` to denote that there exists an assignment which satisfies F, `UNSAT(F)` if there does not, and $v \vDash F$ to denote that v is a satisfying assignment for F – i.e., `SAT(F)` iff $\exists v.v \vDash F$, and otherwise `UNSAT(F)`.

2.1.2 Stochastic Local Search

There are various algorithms for solving SAT instances. The most prominent of the sound but not complete solvers are the *Stochastic Local Search* (SLS) solvers (Nghia Pham et al. 2008). These perform a local search, taking an initial assignment of variables and counting how many clauses this assignment satisfies. Variable's polarities are then flipped to create a new assignment, and if this new assignment does not decrease the number of satisfied clauses this assignment is taken and the procedure is repeated. This is repeated until all clauses are satisfied. Various modifications to this procedure exists, including periodically restarting the search with a different assignment, and weighting clauses differently and dynamically, so satisfying hard to satisfy clauses is rewarded.

2.1.3 Conflict Driven Clause Learning (CDCL)

Conflict Driven Clause Learning (CDCL) (Silva and Sakallah 1996; Marques-Silva and Sakallah 1999; Bayardo and Schrag 1997) is a sound and complete algorithm for solving SAT instances, and is the algorithm used by SAT solvers in this thesis. If a formula is satisfiable, then every variable in every unit clause must be assigned so that each unit clause is satisfied. The repeated application of this rule is known as *unit propagation*. This is a technique crucial to the performance of CDCL. CDCL operates on a Boolean formula as per Algorithm 1.

Algorithm 1: Conflict Driven Clause Learning

Input: Formula F
Output: *True* if F has a satisfying assignment, *False* otherwise

```
1  decisions ← []
2  learnt_clauses ← True
3  while True do
4      if unit propagation causes conflict: F ∧ learnt_clauses ∧ decisions then
5          if decisions = [] then
6              return False
7          clause ← generate_conflict_clause(F, learnt_clauses, decisions)
8          learnt_clauses ← learnt_clauses ∧ clause
9          level ← assertion_level(decisions, clause)
10         decisions ← the first level elements of decisions
11     else
12         if decisions contains all propositions then
13             return True
14         if restarting then
15             decisions ← []
16         literal ← select_literal(F, learnt_clauses, decisions)
17         decisions ← decisions + [literal]
```

CDCL operates with a stack of *decisions*, which are literals representing the current search path. CDCL periodically picks a new decision literal to continue its search, and while this can be done randomly, many methods exist which decrease the expected runtime in practice. Throughout the execution of CDCL, when the decision stack leads to a conflict, a *learnt clause* is derived and used to constrain future searches. This learnt clause represents the "reason" the conflict was encountered. Many techniques exist to find small reasons, with the most trivial being to simply take the disjunction of the negations of all the decision literals.

CDCL initially has an empty decision stack, and an empty set of learnt clauses (Lines 1 and 2). Unit propagation is performed on the conjunction of the original

formula, the decisions literals and the learnt clauses (Line 4). If this causes a conflict, and if there are no decisions constraining the search, then the original formula is unsatisfiable (Line 6). If there are decisions, then a conflict clause is generated and added to the set of learnt clauses (Lines 7 and 8). Along with the conflict clause, the *decision level* of the conflict is also calculated. This is the height at which the decision stack would no longer result in the conflict being encountered. The decision stack is then reduced down to this height to continue the search (Lines 9 and 10).

If the unit propagation does not lead to a conflict, then if all propositions are represented in the decision stack, then this is a valid assignment of variables and so the formula is satisfiable (Line 13). If *restarting* is enabled, then whenever CDCL arrives at a conflict, the decision stack is reset (Line 15). This is very important for efficiency in practice. Then, a new decision literal is chosen and added to the decision stack and the procedure repeats, performing unit propagation again (Lines 16 and 17).

Since the inclusion of conflict clauses does not modify the solution space, they can be removed without altering the correctness of the algorithm. Often in practice, the cost of managing some conflict clauses outweighs their benefits. Schemes exists to delete conflict clauses to try counter this, for instance by trying to estimate their usefulness by their size and/or frequency of use. CDCL is an improvement on the earlier *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm, with the difference that DPLL does not use conflict clauses, and does not restart (Davis et al. 1962).

2.1.4 Incremental SAT

After a SAT solver has solved a SAT instance, it has had the ability to learn much about the problem. For instance, it accumulates conflict clauses in the case of CDCL, or information about good calibrations for heuristics. If the SAT solver is then going to solve another "similar" SAT instance, much of this knowledge can be used to increase the efficiency of solving this second instance. This is known as *incremental* SAT solving (Eén and Sörensson 2003; Gocht and Balyo 2017).

IPASIR (Reentrant Incremental Sat solver API, in reverse) is a standardised interface for incremental SAT solving (Balyo et al. 2016). Through the IPASIR interface, a running SAT instance is maintained. This instance can be repeatedly modified then resolved. One of the ways it can be modified is through the use of *assumptions*. These are unit clauses that only hold true for the next time the running SAT instance is solved. In practice, an underconstrained SAT instance can be established which represents a system, and assumptions can be turned on/off in order to activate

certain parts of the system. Throughout this thesis, we use a SAT instance to represent a small transition system, and repeatedly re-solve it with different assumptions representing different starting states of the system.

After solving a SAT instance and finding it to be unsatisfiable, an incremental SAT solver also provides a set of *used assumptions*. These are a subset of the given assumptions, which would give the same unsatisfiable result – i.e. – for a formula F, assumptions a and used assumptions u, $\texttt{UNSAT}(\texttt{a}\wedge\texttt{F}) \Rightarrow \texttt{UNSAT}(\texttt{u}\wedge\texttt{F})$ and $u \subset a$.

Through the IPASIR interface clauses can be added, but not removed from the solver. However, while clauses cannot be removed via the interface, they can be in practice through the use of *activation literals*. When adding a clause c, instead of adding the clause directly, the disjunction of the clause with the negation of an activation literal a ($\neg a \vee c$) is sometimes added instead. This way, if a is set *True* as an assumption then the added clause effectively becomes c, and if a is set *False*, then the clause is satisfied, effectively removing/deactivating c.

2.2 Classical Planning

The classical planning problem is the problem of determining if there exists a sequence of actions which transitions a certain type of system from a starting state to a state satisfying some goal condition. It is equivalent to a pathfinding problem over a directed graph where nodes in the graph represent states and arcs represent transitions. This is also equivalent to the verification of safety properties in deterministic model checking.

The classical planning problem is given by a tuple $\langle X, A, I, G\rangle$, where:

- X is a set of Boolean-valued state facts known as *propositions*.
- A is a set of *actions*.
- I is the *initial state*.
- G is the *goal condition*.

We overload the term proposition to refer to both planning facts and SAT variables, where the context and use dictates which term we mean. Additionally, there is often a one-to-one correspondence between the two notions.

A *full* problem state can be described by a *cube*, a conjunctive clause, and specifically a conjunction containing exactly one literal for each element in X. A full state is a representation of a total truth assignment to X. The concept of a partial state is also useful, which can be described by a cube over a subset of X – i.e., a representation of

a partial assignment over X. The initial state I is a full state and the goal condition G is typically a partial state.

Each action $a \in A$ is specified in relation to a partial or full state s, by a *precondition* formula $pre(a)$ and an *effect* formula $\mathit{eff}(a)$. We restrict our attention to precondition and effect formulae that are consistent cubes – i.e., if literal ℓ appears in such a cube, then we cannot have $\neg\ell$ also in that cube. For a literal l and a consistent cube c, we use the set notation $l \in c$ to denote that l is a component of c.An action a is *applicable* in a state s iff $\mathtt{SAT}(\mathtt{s} \wedge pre(\mathtt{a}))$. The state $t = succ(s, a)$ resulting from executing a at s satisfies: $\mathtt{SAT}(\mathtt{t} \wedge \mathit{eff}(\mathtt{a}))$ and for every $f \in X$ absent from $\mathit{eff}(a)$ we have that for all $\ell_f \in \{f, \neg f\}$ if $\mathtt{SAT}(\mathtt{s} \wedge \ell_f)$ then $\mathtt{SAT}(\mathtt{t} \wedge \ell_f)$. In other words, an action is applicable to a state if its precondition is consistent with the state; and the successor state will be the same as the original state, except that all variables mentioned in the effect are made to be consistent with the effect. In the case of a partial state, the successor may not be fully determined, and should be taken to be any one total state satisfying the above condition.It is important that applicability is defined for partial states, as it is important for the efficiency of *Property Directed Reachability*, introduced in Section 2.5. An action *a conflicts* with another a' if $\mathtt{UNSAT}(\mathit{eff}(\mathtt{a}) \wedge \mathit{eff}(\mathtt{a}'))$. Those two actions *interfere* if either $\mathtt{UNSAT}(\mathit{eff}(\mathtt{a}) \wedge pre(\mathtt{a}'))$ or $\mathtt{UNSAT}(\mathit{eff}(\mathtt{a}') \wedge pre(\mathtt{a}))$ – i.e., if the preconditions and effects of the two actions are inconsistent.

Example 4 (Representing a Planning Problem)
As an example we represent the Logistics problem from Example 1 formally as a planning problem. The domain has two locations, $L1$ *and* $L2$*, two packages,* $P1$ *and* $P2$*, and one truck* T*. Packages and trucks, can be at locations, and packages can be in trucks. There is one proposition for each object-location setting – e.g.,* $at(P1, L1)$ *representing that package* $P1$ *is at location* $L1$*. Similarly, there is a proposition for each package in each truck – e.g.,* $in(P1, T)$ *is a proposition representing that package* $P1$ *is in truck* T*. An object can only be in one place at a time – e.g.,* $at(P1, L1) \wedge at(P1, L2)$ *is not a fragment of a valid state. Actions specify that a truck can drive from any location to any other location, and that packages can be loaded in or out of a truck at a truck's current location. The planning problem is then given as* $\langle X, A, I, G\rangle$ *where:*

$$\begin{aligned} X = \; & \{at(P1, L1), at(P1, L2), in(P1, T) \\ & at(P2, L1), at(P2, L2), in(P2, T) \\ & at(T, L1), at(T, L2)\} \end{aligned}$$

The set of actions A *is given by:*

action	pre(*action*)	eff(*action*)
$drive(T, L1, L2)$	$at(T, L1)$	$\neg at(T, L1) \wedge at(T, L2)$
$drive(T, L2, L1)$	$at(T, L2)$	$at(T, L1) \wedge \neg at(T, L2)$
$unload(T, P1, L1)$	$in(P1, T) \wedge at(T, L1)$	$at(P1, L1) \wedge \neg in(P1, T)$
$unload(T, P2, L1)$	$in(P2, T) \wedge at(T, L1)$	$at(P2, L1) \wedge \neg in(P2, T)$
$unload(T, P1, L2)$	$in(P1, T) \wedge at(T, L2)$	$at(P1, L2) \wedge \neg in(P1, T)$
$unload(T, P2, L2)$	$in(P2, T) \wedge at(T, L2)$	$at(P2, L2) \wedge \neg in(P2, T)$
$load(T, P1, L1)$	$at(P1, L1) \wedge at(T, L1)$	$\neg at(P1, L1) \wedge in(P1, T)$
$load(T, P2, L1)$	$at(P2, L1) \wedge at(T, L1)$	$\neg at(P2, L1) \wedge in(P2, T)$
$load(T, P1, L2)$	$at(P1, L2) \wedge at(T, L2)$	$\neg at(P1, L2) \wedge in(P1, T)$
$load(T, P2, L2)$	$at(P2, L2) \wedge at(T, L2)$	$\neg at(P2, L2) \wedge in(P2, T)$

$$\begin{aligned} I = \quad & \{at(P1, L1), \neg at(P1, L2), \neg in(P1, T) \\ & at(P2, L1), \neg at(P2, L2), \neg in(P2, T) \\ & at(T, L1), \neg at(T, L2)\} \end{aligned}$$

$$G = \quad \{at(P1, L2), at(T, P2)\}$$

The *Stanford Research Institute Problem Solver* (STRIPS) was a solver for classical planning, which also defined a standard representation now also known as STRIPS (Fikes and Nilsson 1971). STRIPS problems, as represented in the *Planning Domain Definition Language* (PDDL) are a standard representation for these problems, and is the format we assume for the purposes of this thesis (McDermott et al. 1998).

2.2.1 Mutual Exclusion (Mutex) Invariants

We make use of *invariants*, which in general are formulae that hold true in the initial state, and all states reachable from the initial state (Gerevini and Schubert 1998; Rintanen 2008). We shall use *mutex* invariants, of the form $\neg f_1 \vee \neg f_2$, in particular. That disjunctive clause says that one of the proposition f_1 or f_2 must be *False*. Sets of useful mutex invariants are routinely supplied by preprocessing routines in planning tools, such as MADAGASCAR (Rintanen 2012). These are not necessary for correctness, but are very useful for efficiency in practice.

As an example consider the Logistics domain. A package can only be at one location, so the invariant of the form $\neg at(P, L1) \vee \neg at(P, L2)$ could be generated for all packages and pairs of locations. Note the common SAS+ representation uses multi-valued state variables, with mutex invariants between each of the possible values (Bäckström and Nebel 1995).

2.3 Fully Observable Non-Deterministic (FOND) Planning

Fully Observable Non-Deterministic (FOND) planning is a class of decision-making problems characterized by the presence of non-deterministic (unpredictable) effects of actions and full observability, that is, the ability to always uniquely determine the current state during the execution of a plan. The initial state of the system is uniquely defined, and the objective of a plan is to reach one of a set of designated goal states. This problem is related to the (fully observable) Markov Decision Process (MDP) problem (Puterman 1994), with two main differences, the absence of probabilities for the different possible outcomes of an action, and the absence of rewards. In Chapter 6 we propose a variation of PDR to solve FOND instances.

In classical planning, when an action is executed, a specific state is deemed the successor. However in FOND planning, when an action is executed, it has a set of *outcomes*, each of which may be the successor state. Which outcome is chosen as the successor is done nondeterministically. For the purposes of this thesis, any probabilities associated with which outcome is to be chosen is irrelevant, as long as they are all non-zero. We operate under the *fairness assumption*, that if a nondeterministic action is taken repeatedly on a state, the number of applications needed to end up at a given outcome will always be finite. I.e. if an action is repeatedly applied to a state, each outcome will eventually be encountered, and after that encounter, will eventually be encountered again, and so forth (Cimatti et al. 2003).

Like in classical planning, a FOND planning task is given by a tuple $\langle X, A, I, G\rangle$, where X is a set of Boolean-valued state propositions, A is a set of actions, I is a total assignment for X describing the initial state, and G is the goal condition. We represent a state s as a truth-value assignment to all elements of X. A state s is a goal state iff $s \models G$. Unique to FOND, each action $a \in A$ is described by a precondition formula $pre(a)$ and a set of outcomes $\mathcal{O}(a) = \{o_1, .., o_k\}$, each of which describes an effect $\mathit{eff}(o_i)$. An outcome effect $\mathit{eff}(o_i)$ is a consistent set of literals over X. When a is executed, nature decides the outcome in $\mathcal{O}(a)$ that occurs in such a way that the property of fairness is always provided.

An action a is executable at state s iff $s \models pre(a)$. The state $s' = succ(s, o)$ resulting from an outcome o selected by nature in this scenario satisfies the following: $s' \models \mathit{eff}(o)$, and also for every $f \in X$ absent from $\mathit{eff}(o)$ we have $s \models f$ iff $s' \models f$. The set of successor states of s with respect to action a is $succ(s, a) = \{succ(s, o) | o \in \mathcal{O}(a)\}$.

Following (Martelli and Montanari 1978) and more recently (Hansen and Zilberstein

1998), we represent problem states, actions and information about how states are reached using actions as a directed F(orward)-hypergraph (V, E), where V are vertices in correspondence with problem states and E are k-connectors/hyperarcs each representing an action and that action's possible outcomes. We adopt the convention that vertices and hyperarcs are labelled with their corresponding state and action, respectively. Each f(orward)-arc $(v, V') \in E$ is an ordered pair where v corresponds to a state from which an action a can be executed, and V' all successors of v under the label a of the arc. Characterising reachability, a vertex is reachable from itself and otherwise v' is reached from v if $(v, V') \in E$ and $v' \in V'$. A hyperpath from v_0 to v_n is a sequence of hyperarcs $(v_0, V_1), (v_1, V_2) \ldots, (v_{n-1}, V_n)$ where $v_i \in V_i$ for all $i \in 1..n$.

The solution concept we seek is that of a *strong cyclic policy* under the fairness assumptions – herein referred to a policy (Cimatti et al. 2003).

A policy denoted π, is an assignment of actions to states, and corresponds to a subgraph of the concrete problem graph just described above, where the only states which do not have outgoing arcs are goal states. Furthermore, for it to be a policy for a state s, by starting at s, and repeatedly following the prescribed actions, one must be guaranteed to eventually transition to a goal state (at which point the execution stops), as long as the chosen outcome from each action abides by the fairness assumptions. For a policy π, we write $\lceil \pi \rceil_s$ to denote the maximum possible number of distinct states encountered in a trace from s to the goal (not including s). For example, consider we have n fair coins on a table and the only available action is to thump the table simultaneously flipping all coins fairly. Suppose both the goal and starting states are total assignments to the states of the coins. The only available policy has $\lceil \pi \rceil_I = 2^n - 1$ whenever the starting state does not equal the goal, and is otherwise 0.

2.3.1 Related Work

There are logic-based approaches to solving the FOND planning problem, including *FOND-SAT* in which a single SAT call is used to determine the existence of a graph that represents all possible executions of a successful plan that always reaches the goal (Geffner and Geffner 2018). Earlier, not as well scalable approaches to FOND planning include methods based on BDDs and symbolic breadth-first search (Cimatti et al. 1998, 2003), and the use of quantified Boolean formulae (QBF) (Rintanen 1999; Turner 2002). The works by Cimatti et al. introduced the central concept of strong cyclic plans, broadly used in later works.

Algorithms that use explicit state-space search in AND-OR graphs have been used

(Mattmüller et al. 2010; Ramirez and Sardina 2014), as well as algorithms that call classical (deterministic, single-initial-state) planning as a subroutine for finding execution paths to goal states (Muise et al. 2012, 2024). While these approaches fare very well on many of the existing benchmark sets, (Geffner and Geffner 2018) have argued that this is at least partly due to simple features of those benchmarks, allowing them – to an extent – to effectively ignore non-deterministic alternative outcomes.

2.4 SAT Planning

SAT Planning is a term used to describe a variety of techniques for solving the planning problem via SAT (Kautz and Selman 1992; Biere et al. 2009; Rintanen 2012). This is usually done by taking a series of bounded versions of the planning problem with increasing bounds, converting these to SAT, and solving them until a solution is found. Many such conversions exist. For this thesis, we focus on encodings which bound the number of planning steps, and where SAT variables directly correspond to whether a proposition is *True* in a time-step and whether an action is executed in a time-step.

To denote a proposition p is *True* or an action a is executed at an initial/originating time-step we simply write these variables as is. To denote these in the successor time-step we write p' and a'. It is important for the context of this thesis to note that these time-steps are relative. By default we do not write p to denote time-step one and p' to denote time-step two, rather we write p to be an originating time-step and p' to be the successor. If referring to the n'th step, it is only the n'th step relative to the originating time-step. Similarly for a formula F, possibly containing many propositions/actions in different time-steps, we write F' to denote a copy of the formula where each proposition/action refers to that proposition/action in the successive time-step. We also use superscripts to denote further successive steps. A useful advantage of SAT planning is that preconditions and effects can be extended to arbitrary formula, possibly spanning multiple time-steps.

Throughout this thesis we use the $\forall$-*step* encoding (Rintanen et al. 2006), however many other encodings exist (Kautz et al. 1996; Rintanen et al. 2006; Robinson et al. 2008).

2.4.1 $\forall$-Step Encoding

We describe the $\forall$-step encoding of bounded planning into SAT. The $\forall$-step encoding allows multiple actions to be scheduled inside a single planning step, as long as the

actions within any step can serialized in any order without affecting the outcome. We call the encoding ∀-step because a valid plan is created for all serializations of the parallel plan. This can be guaranteed if all actions in a planning step do not conflict or interfere with each other. Allowing multiple actions within a planning step gives more flexibility, makes SAT instances smaller, and tends to improve performance.

We break the encoding into three parts:

- $\mathcal{I}$: Constraining a step to be consistent with the initial state,
- $\mathcal{G}$: Constraining a step to be consistent with the goal condition,
- $\mathcal{T}$: The transition function, defining the rules for how one state transitions to the next.

We will describe the contents of each of these formulae, but first show how they can be used together. A SAT instance representing a planning problem bounded to n planning steps is represented as:

$$\mathcal{I} \wedge \bigwedge_{k=0}^{k=n-1} (\mathcal{T}^k) \wedge \mathcal{G}^n$$

If the SAT instance is satisfiable, then to extract a parallel plan, the set of *True* action variables for each time-step is extracted from a satisfying valuation of this formula. To create a serial plan, the parallel actions within each time-step can be ordered arbitrarily.

The formula $\mathcal{I}$ corresponds directly to the initial state, for every *True*/*False* proposition there is a matching *True*/*False* proposition in the formula. $\mathcal{G}$ is similar, corresponding to the goal condition, but with the addition of the invariant constraints. The addition of these constraints does not influence the correctness, as all possible goal states will also adhere to the invariant constraints, by the definition of the invariants. We define $\mathcal{G}$ in this way to ease in the presentation of the correctness of PDR (Lemma 5).

$\mathcal{T}$ is more complicated and composed of multiple parts given in Schemas 1-5.

Schema 1 (Action Implies Precondition)
For each action, we impose that it can only be executed in a time-step if its preconditions are met. For each action $a \in A$, for each proposition in its precondition $p \in \text{pre}(a)$ we have:

$$a \Rightarrow p$$

■

Schema 2 (Action Implies Effect)
Similarly, if an action is executed at a time-step, then the successor state must be consistent with its effects. For each action $a \in A$, for each proposition in its effect $e \in \text{eff}(a)$ we have:

$$a \Rightarrow e'$$

■

Schema 3 (Frame Axioms)
The propositions mentioned in the effects of executed actions will be constrained, but conversely, we need to enforce that propositions that are not mentioned in the effects of executed actions do not change polarity. The clauses which enforce this are known as frame axioms. *For each proposition $p \in X$ we have:*

$$p' \Rightarrow p \vee (a_{p,1} \vee ... \vee a_{p,m})$$

Where $a_{p,1}...a_{p,m}$ are all the actions which have p in their effects. Similarly we also have:

$$\neg p' \Rightarrow \neg p \vee (a_{\neg p,1} \vee ... \vee a_{\neg p,m})$$

Where $a_{\neg p,1}...a_{\neg p,m}$ are all the actions which have $\neg p$ in their effects. ■

Note that this Schema works in tandem with Schema 2. For instance if p holds, then an action is executed which turns p false in the next time step ($\neg p'$ holds), then this result is enforced in schema 2, not this Schema.

Schema 4 (Interfering Actions Mutex)
The $\forall$-step semantics require actions executed in the same time-step to be neither conflicting nor interfering. Conflicting actions are banned from occurring together implicitly, but interfering actions need to be explicitly banned. These are expressed as mutex clauses over actions, and are included in $\mathcal{T}$. ■

Schema 5 (State Mutex Invariants)
Also included in $\mathcal{T}$ is the set of invariants. These binary clauses are used to constrain both the originating step, and the successor step. These are not necessary for correctness, but important for efficiency. ■

2.4.2 Completeness Threshold

The *horizon* is the number of time-steps where an action is applicable, so one less than the number of time-steps represented in the SAT instance. If a plan exists with this many steps then this method will find it. However if a planning instance has no solution, then neither will any bounded version. By default, by only testing horizon-bounded versions of a problem a solver cannot deduce that a plan does not exist

globally, only that one doesn't exist for the tested horizons. This can be mitigated with the use of a *completeness threshold*. This is a horizon where, if a plan does not exist at this horizon then a plan does not exist for the original problem. A trivial completeness threshold is $(2^{|X|} - 1)$ – i.e. – giving the solver enough time-steps to go through every possible state – if a plan cannot be found under this horizon then one never will be. There exists methods for finding smaller completeness thresholds, however these are often still too large to be used in practice. Some methods utilize the dependency structure of the problem to break the problem into smaller parts, then creating a completeness threshold for the concrete problem by combining the thresholds of component parts (Abdulaziz and Berger 2021). Some decomposition methods are explored later in this chapter (Section 2.6).

2.4.3 Query strategies

In the case that a plan exists it is not usually known beforehand how many steps will be required. There is then a question of what horizon to use. Too small and a plan will not be found. Too large and the SAT solver can get "lost in the weeds" spending excess time and memory searching for a plan. Often, a collection of SAT calls at different horizons are used, and there exist various query strategies for this (Streeter and Smith 2007; Rintanen 2012). The most simple strategy being to query if a plan exists when the horizon is one, if it does not then query if a plan exists when the horizon is two, then three, then four and so on. More advanced query strategies may query horizons in a geometric sequence (e.g. $\{1, 2, 4, 8, 16...\}$), and/or divide up processing time so multiple horizons are queried at once, with more processing power being spent on lower horizons.

2.4.4 Related Work

Planning via SAT was first introduced by Kautz and Selman in their seminal 1992 paper (Kautz and Selman 1992; Kautz et al. 1996). They used their *GSAT* SLS SAT solver as their underlying SAT solver (Selman et al. 1992; Selman and Kautz 1993). Being an SLS solver, it could not prove plan non-existence. In 2004, an extension to this work, *SatPlan*, won first place in the International Planning Competition optimal deterministic planning track (Kautz et al. 2006). This solver used the sound and complete CDCL solver *Siege*, and as such could prove plan non-existence for set horizons. In 2012, Rintanen developed *Madagascar*, a SAT based planning procedure which incorporated many efficiency features, including parallel query strategies and the $\exists$-*step* semantics, allowing more actions to be scheduled per time-step (Rintanen 2012). Of particular note is the inclusion of a bespoke SAT solver designed

specifically to solve encodings of planning problems.

Many different encodings have been proposed for translating planning problems into SAT problems. In the ∀-step encoding, a SAT variable must exist for each action, representing each possible combination of parameters. For instance in a Logistics problem there must be a corresponding action for each truck, origin and destination combination. If there were 10 trucks and 10 fully connected cities, there would be $10 * 10 * 9 = 900$ actions. In 2009 Robinson et al. present an encoding which bypasses this issue by splitting the representation of the actions. In a naive splitting approach, a SAT variable exists to represent whether any drive action is executed at all, and a separate SAT variable exists for each option of each parameter. For the above example, this encoding would result in only $1 + 10 + 10 + 10 = 31$ SAT variables being needed for the representation of these actions.

When solving a horizon n bounded planning problem via SAT, the resulting SAT instance is often quite similar to that of the horizon $n + 1$ bounded instance. *Incplan* is a SAT planner which uses incremental SAT solvers to exploit this (Gocht and Balyo 2017). Various methods are implemented, but the most effective is to "break" the middle time-step, and replace it with two time-steps. This is done via an extensive use of activation literals to (de)activate certain clauses. Learnt clauses which include propositions close to the first and last time-step are unlikely to also include propositions from the middle time-step. Because of this, as only the middle time-step is modified, this knowledge can be retained between successive SAT calls, accelerating the search.

2.5 Property Directed Reachability

We here present *Property Directed Reachability* (PDR), an algorithm key to this thesis.

2.5.1 Overview and Preamble

PDR is a sound and complete SAT based algorithm for solving the planning problem (Bradley 2011; Suda 2014). There are various presentations of PDR, we largely follow the presentation of (Suda 2014) as it works well for the planning setting. PDR proceeds by maintaining and iteratively refining overapproximations of reachability information, called *layer* information about the system. The information at each layer is represented as a CNF formula. A layer formula $\mathcal{L}_i$ is associated with each discrete step i. Each formula $\mathcal{L}_i$ satisfies an invariant condition: For every state s such that $s \not\models \mathcal{L}_i$ there is no ∀-step execution of length i or less from s to a

state satisfying G. In other words, if a state does not satisfy the layer formula at step i, then it is known that that state cannot reach the goal in i or less steps. Initially, $\mathcal{L}_0 = G$ and all other layers are the vacuously satisfied empty CNF formula containing no clauses – i.e., the loosest possible overapproximation. If a clause is added to a layer at index i, then it is always included at the layers of all lower indicies as well. In other words, the layer formulae progressively become looser at higher indices. In practice, instead of storing all clauses contained in each layer, just the difference – the clauses in the layer not in the lower layer are stored. A "*queue*", Q, is maintained with each element a tuple $\langle s, i \rangle$ known as an *obligation*, where s is a state and i is a layer index. Each obligation $\langle s, i \rangle \in Q$ represents the question: Can s reach the goal in i steps? We say a state s is *at* layer i in the queue if the obligation $\langle s, i \rangle$ is in the queue.

PDR proceeds by taking an obligation $\langle s, i \rangle$ from the queue, and querying whether or not there exists a successor state t which satisfies the layer formula $\mathcal{L}_{i-1}$. In our work that query is evaluated using a general purpose incremental SAT solver. Specifically, the SAT formula contains the state s associated with an obligation at i, the successive layer formula $\mathcal{L}_{i-1}$, and the transition formula $\mathcal{T}$. A single incremental SAT solver can be used for each layer throughout the execution of PDR. The solver initially contains $\mathcal{T}$. Whenever a clause is added to $\mathcal{L}_{i-1}$, this can be added to the solver, and whenever a SAT call is to be made, the state s can be represented as a collection of assumptions. In this way, the same incremental SAT solver instance can be used many times on slightly different problems, which is important for efficiency. This SAT problem then has a solution iff s has a successor t such that $t \vDash \mathcal{L}_{i-1}$. If that formula is satisfiable then the successor state t is extracted from the model. The original obligation $\langle s, i \rangle$ and the successor obligation $\langle t, i-1 \rangle$ are both then added to the queue. Note we add the original obligation back to the queue because it was removed from the queue earlier.

If no successor state exists, then a *reason* is derived and added to $\mathcal{L}_i$. A reason, r is a partial state consistent with s, where r also does not have a successor state satisfying $\mathcal{L}_{i-1}$. The clause $\neg r$ is then added to $\mathcal{L}_i$. After conjoining, for any state y where $\mathtt{SAT}(\mathtt{y} \wedge \mathtt{r})$ we have $\mathtt{UNSAT}(\mathtt{y} \wedge \mathcal{L}_\mathtt{i})$. The state s is itself a reason, but adding $\neg s$ to a layer formula is not effective. By finding a small reason with fewer literals, the added reason is a relatively strong constraint. Our use of incremental SAT accelerates the process of finding good reasons. In order to generate the reason r, PDR begins with the partial state u indicated by the used assumptions, as this will be a valid reason. This candidate reason is further tightened, by further queries to an incremental SAT procedure. Specifically, we iteratively test whether each literal can be removed from

r so that the obligation $\langle r, i\rangle$ remains unprogressable. A literal is tested for removal by using an incremental SAT solver to determine if the reason without that literal is still valid. Where the obligation $\langle r, i\rangle$ is unprogressible, the corresponding used assumptions are taken forward as the revised reason. This process continues until every literal has been tested once for removal, or is otherwise removed by being excluded for not being a used assumption.

All states mentioned in queued obligations are either the initial state, or a state that can be reached from the initial state. As such, if at any point a goal state is to be added to the queue – i.e. a state as part of an obligation at layer zero – then the goal is reachable from the initial state, and a plan exists. Through auxiliary accounting of which actions were taken to transition from the initial state to the goal state, a plan can be extracted.

2.5.2 The Core Algorithm

We can now tie these ideas to the pseudocode of Algorithm 2. First, the queue and layer information are initialised (Line 1). A check is made to see if the initial state satisfies the goal condition, and if so then the problem is trivially satisfiable (Line 2). After that, a loop over the length of planning horizons $k \in [1, 2, 3, ...]$ commences (Line 3). When the algorithm comes to increment k without having found a plan, then $\texttt{UNSAT}(\texttt{I} \wedge \mathcal{L}_{\texttt{k}})$, and therefore there is no plan with k or fewer steps. At the beginning of iteration k the obligation $\langle I, k\rangle$ is pushed to Q, and then a loop starting on Line 5 is entered which is broken once the queue is empty. This indicates that work will continue at k until there are not obligations on Q to evaluate. The loop starts by taking an obligation $\langle s, i\rangle$ off the queue (Line 6). It is important that an obligation with minimal i – i.e. a depth first approach – is taken, as otherwise there is nothing to stop the initial state from repeatedly being taken off the queue and progressed to the same successor obligation forever. A SAT call is made to determine if s has a successor which satisfies $\mathcal{L}_{i-1}$ – i.e. that is one step closer to the goal (Line 7).

If a successor state t exists, it is extracted from the satisfying model resulting from the SAT call, and added to the queue as an obligation at the successive layer $(i-1)$ along with the originating obligation (Lines 8, 11). If the successor state is a goal state, then *True* is returned as a goal state is reachable from the initial state.

However if a successor state cannot be found then a reason r is generated from s by iteratively calling an incremental SAT procedure as described above (Line 13). This reason is then used to further constrain the layer information at, and all layers lower than, i (Line 15). *Obligation rescheduling* is then performed. This is a procedure

Algorithm 2: Property Directed Reachability

Input: A planning problem $\langle X, A, I, G\rangle$

Output: *True* if the problem is solvable, *False* otherwise

$Q \leftarrow \{\}, \mathcal{L}_0 \leftarrow G$, for $j > 0 : \mathcal{L}_j \leftarrow \mathit{True}$
if $I \vDash \mathcal{L}_0$ **then return** True
for $k \in [1, 2, 3, ...]$ **do**
  $Q \leftarrow \{\langle I, k\rangle\}$
  while $Q \neq \{\}$ **do**
    $\langle s, i\rangle \leftarrow$ pop most recently added $\langle s, i\rangle$ from Q with minimal i.
    if $\mathtt{SAT}(\mathtt{s} \wedge \mathcal{T} \wedge \mathcal{L}'_{\mathtt{i-1}})$ **then**
      $t \leftarrow$ extract the successor state from the satisfying model
      if $i - 1 = 0$ **then**
        return True
      $Q \leftarrow Q \cup \{\langle s, i\rangle, \langle t, i - 1\rangle\}$
    else
      $r \leftarrow$ generate a reason from s
      foreach $j \in \{0, ..., i\}$ **do**
        $\mathcal{L}_j \leftarrow \mathcal{L}_j \wedge \neg r$
      if $i < k$ **then**
        $Q \leftarrow Q \cup \{\langle s, i + 1\rangle\}$ // Obligation Rescheduling
      // Trim the queue with the new reason
      $Q_{trimmed} \leftarrow \{\}$
      foreach $\langle z, h\rangle \in Q$ **do**
        if $h > i$ *or* $\mathtt{UNSAT}(\mathtt{r} \wedge \mathtt{z})$ **then**
          $Q_{trimmed} \leftarrow Q \cup \{\langle z, h\rangle\}$ // Keep as is
        else if $i < k$ **then**
          $Q_{trimmed} \leftarrow Q \cup \{\langle z, i + 1\rangle\}$ // Keep at a higher layer index
      $Q \leftarrow Q_{trimmed}$
  for $i \in \{1, .., k + 1\}$ **do** /* Clause Pushing */
    foreach *clause* $c \in \mathcal{L}_{i-1}, c \notin \mathcal{L}_i$ **do**
      if $\mathtt{UNSAT}(\neg\mathtt{c} \wedge \mathcal{T} \wedge \mathcal{L}'_{\mathtt{i-1}})$ **then**
        $\mathcal{L}_i \leftarrow \mathcal{L}_i \wedge c$
    if $\mathcal{L}_i \equiv \mathcal{L}_{i-1}$ **then**
      return False

not necessary for correctness, but is very useful for efficiency. If a state cannot be progressed at layer i, and $i < k$, then a new obligation is added for this state at $i+1$. This effectively "gives the state another chance", it was not able to be progressed at i, perhaps it can be progressed at the possibly less constrained $i + 1$. The queue is then trimmed so that every state which is now inconsistent with the newly added clauses is either pushed back to a layer where the state is consistent with the layer formula, or if the state is already at layer k, taken off the queue (Lines 19-25). Note if obligation rescheduling/queue trimming is disabled, plans found will be of minimum length.

Once the queue is empty, then *clause pushing* is performed along with a test for termination. For each $i \in \{1, .., k + 1\}$, clause pushing involves considering each clause in $\mathcal{L}_{i-1}$ and testing if it would be a valid clause to constrain $\mathcal{L}_i$, and if so, adding it there (Lines 26-29). Clause pushing is performed for performance reasons, and like obligation rescheduling, is not required for completeness. To test for termination, the algorithm checks if two consecutive layers are equivalent. If they are then the problem has no solution, so *False* is returned (Line 31). Note this equivalence check is a syntactic check, of whether the set of clauses in adjacent layer formulae are identical. As $\mathcal{L}_i \subset \mathcal{L}_j, i \leq j$, instead of storing the set of clauses in a layer directly, only the difference between layers is stored. As such, checking if two consecutive layers are identical is cheap as it is simply checking if the difference set is empty.

An example execution of PDR is shown in Example 5.

Example 5 (PDR Execution on an Example Problem)
We demonstrate the execution of PDR on the running Logistics problem presented in Example 1/4. Initially $\mathcal{L}_0$ *is constrained to be equal to the goal condition (Line 1). If the initial state is not consistent with the goal state we place the obligation* $\langle I, 1\rangle$ *on the queue. We present the state of the layer formulae and queue in the following table. For ease of presentation, we only list the positive propositions when listing states*[1]*: Each line in a cell either represents a clause as part of a layer formula, or a state at an index in the queue.*

Index	0	$k = 1$
$\mathcal{L}$	$at(P1, L2)$ $at(T, L2)$	
Q		$at(P1, L1) \wedge at(P2, L1) \wedge at(T, L1)$

Next, the obligation $\langle at(P1, L1) \wedge at(P2, L1) \wedge at(T, L1), 1\rangle$ *is taken off the queue to be progressed (Lines 6 and 7). This state has no successor that models* $\mathcal{L}_0$*, so a reason is found (Line 13). As an example,* $at(T, L1)$ *could not be used as a reason, because the truck can drive to* $L2$ *in one action, and the location of* $P1$ *is unspecified, so could be set to be* $L2$*. Multiple different reasons could exist, we here choose the reason* $at(P1, L1)$*.*

Note that while in this example $at(P1, L1)$ *conveys the same information as* $\neg at(P1, L2)$*, in general, denoting that a proposition is* False *in the reason may create a looser formula than denoting that a certain proposition is* True*. This looser reason then creates a tigher – and often more useful – constraint for the layer formula.*

This reason is then used to constrain the layer formula (Line 15). $\mathcal{L}_0$ *is already constrained by* $at(P1, L2)$*, so the addition of* $\neg at(P1, L1)$ *does not constrain it any further.*

[1] In practice, with mutex clauses, these positive propositions would imply the negative propositions.

Index	0	$k = 1$
$\mathcal{L}$	$at(P1, L2)$ $at(T, L2)$ $\neg at(P1, L1)$	$\neg at(P1, L1)$
Q		

At this point the queue is empty, so clause pushing is performed which yields no change (Lines 26-29). Then the layer formulae are checked to see if there are consecutive identical layers at or below index k, which there are not (Lines 30 and 31). Now k is incremented, and an obligation for the initial state is added to the queue.

Index	0	1	$k = 2$
$\mathcal{L}$	$at(P1, L2)$ $at(T, L2)$ $\neg at(P1, L1)$	$\neg at(P1, L1)$	
Q			$at(P1, L1) \wedge at(P2, L1) \wedge at(T, L1)$

Then, the obligation $\langle at(P1, L1) \wedge at(P2, L1) \wedge at(T, L1), 2\rangle$ *is taken off the queue to be progressed. A successor which satisfies* $\mathcal{L}_1$ *is found, as* $P1$ *can be loaded into the truck, which would make it not at* $L1$*. The originating obligation, as well as a new obligation for this successor state is added to the queue.*

Index	0	1	$k = 2$
$\mathcal{L}$	$at(P1, L2)$ $at(T, L2)$ $\neg at(P1, L1)$	$\neg at(P1, L1)$	
Q		$in(P1, T) \wedge at(P2, L1) \wedge at(T, L1)$	$at(P1, L1) \wedge at(P2, L1) \wedge at(T, L1)$

This new obligation $\langle in(P1, T) \wedge at(P2, L1) \wedge at(T, L1), 1\rangle$ *is taken off to be progressed. It cannot be progressed so a reason* $in(P1, T) \wedge at(T, L1)$ *is found then used to constrain the layer formula. This causes the originating obligation to be rescheduled to Layer 2.*

Index	0	1	$k = 2$
$\mathcal{L}$	$at(P1, L2)$ $at(T, L2)$ $\neg at(P1, L1)$ $\neg in(P1, T) \vee \neg at(T, L1)$	$\neg at(P1, L1)$ $\neg in(P1, T) \vee \neg at(T, L1)$	
Q			$at(P1, L1) \wedge at(P2, L1) \wedge at(T, L1)$ $in(P1, T) \wedge at(P2, L1) \wedge at(T, L1)$

The rescheduled obligation $\langle in(P1, T) \wedge at(P2, L1) \wedge at(T, L1), 2\rangle$ *is taken back off the queue, successfully progressed (by driving to* $L2$*) to the state* $in(P1, T) \wedge at(P2, L1) \wedge at(T, L2)$*. This successor state forms a new obligation.*

Index	0	1	$k = 2$
$\mathcal{L}$	$at(P1, L2)$ $at(T, L2)$ $\neg at(P1, L1)$ $\neg in(P1, T) \vee \neg at(T, L1)$	$\neg at(P1, L1)$ $\neg in(P1, T) \vee \neg at(T, L1)$	
Q		$in(P1, T) \wedge at(P2, L1) \wedge at(T, L2)$	$at(P1, L1) \wedge at(P2, L1) \wedge at(T, L1)$ $in(P1, T) \wedge at(P2, L1) \wedge at(T, L1)$

This obligation is then progressed again by unloading $P1$ *from the truck, resulting in state* $at(P1, L2) \wedge at(P2, L1) \wedge at(T, L2)$*, which satisfies* $\mathcal{L}_0$*, and as such is a goal state, so PDR returns True (Line 10).*

2.5.3 Correctness

We provide here a proof of correctness following the proofs provided by (Bradley 2011; Suda 2014). Note that Suda's proof requires all reasons to be inconsistent with the goal, which is not required, and our proof does not appeal to that. Additionally, our addition of queue trimming requires a slight variation to the proofs. Also note in Chapter 6 we present a variation of PDR which operates on FOND problems, and includes a proof of correctness which builds upon some of the concepts presented here. The proof is broken down into five lemmas before the main theorem.

Lemma 1 ($\mathcal{L}_i \Rightarrow \mathcal{L}_{i+1}$ for all $j \geq 0$)
Proof. This is true initially as $\mathcal{L}_0 \Rightarrow \mathcal{L}_1$ ($\mathcal{G} \Rightarrow True$) and the remainder of the layer formulae are empty. The layer clauses are only modified in two places, Lines 15 and 29. in both these places, clauses are only ever added to layers with higher indices if they also exist/are added to all layers with lower indices. ■

Lemma 2 ($\mathcal{L}'_{i-1} \wedge \mathcal{T} \Rightarrow \mathcal{L}_i$ for all $i > 0$)
Proof. This is true initially, as initially $\mathcal{L}_i \equiv True$ for all $i > 0$. When a layer formula at Layer i is modified via a failed obligation, it is only by the addition of the negation of a reason, $\neg r$. By the definition of a reason, $\texttt{UNSAT}(\texttt{r} \wedge \mathcal{T} \wedge \mathcal{L}_{\texttt{i}-1})$ and as such, $\mathcal{L}'_{i-1} \wedge \mathcal{T} \Rightarrow \neg r$. So the addition of this clause to $\mathcal{L}_i$ does not invalidate this invariant.

For a more intuitive explanation of this invariant, the set of states that satisfy $\mathcal{L}'_{i-1} \wedge \mathcal{T}$ can be thought of as all the states that can transition to a state satisfying $\mathcal{L}_{i-1}$ in one step. $\mathcal{L}'_{i-1} \wedge \mathcal{T} \Rightarrow \mathcal{L}_i$ then says that the set of states that can transition to $\mathcal{L}_{i-1}$ in one step are consistent with $\mathcal{L}_i$. This is to be expected, as no state that can transition to $\mathcal{L}_{i-1}$ in one step would be forbidden by $\mathcal{L}_i$. ■

Lemma 3 (When an iteration of k (Line 3) finishes the path construction phase (exits the loop at Line 5), then the initial state is incompatible with $\mathcal{L}_k$)
Proof. At the start of the iteration, the queue contains the obligation $\langle I, k \rangle$. If the iteration has finished, then this state must have been removed from the queue. States are only ever removed from the queue (and not subsequently put back on, as in Line 11) when they are incompatible with the layer they are posted at. Either by being found to be unprogressable at that layer, and a reason is added to the layer to ban it, or by being trimmed from the queue because another reason was added which caused the state to be incompatible with the layer. ■

Lemma 4 (When PDR creates a new obligation $\langle s, i \rangle$, (Lines 4 and 8), then at the time of creation $s \vDash \mathcal{L}_i$. Furthermore, $s \nvDash \mathcal{L}_j$ for $j < i$)

Proof. This lemma states that a created obligation's state is always consistent with the layer at the index it is posted at, and inconsistent with layers with lower indicies – i.e. it is always at the lowest index layer it is consistent with.

Note that for the second part, if it is shown to be true for $j = i - 1$, then by the invariant in Lemma 1 it will be true for all $j < i$. In PDR, clauses are never removed from the layer formula, so we can focus on the addition of obligations as if $s \not\models \mathcal{L}_j$ at any point, then it will forevermore.

This is true at the start of each k-iteration when $\langle I, k\rangle$ is added to the queue, as $\mathcal{L}_k$ starts as the empty clause, and $s \not\models \mathcal{L}_{k-1}$ by the invariant in Lemma 3 (or $k = 1$, so $\mathcal{L}_{k-1}$ is equivalent to or tighter than $\mathcal{G}$).

We will assume that the invariant condition holds, and show that it then continues to hold after the creation of a new obligation. Consider an obligation $\langle s, i\rangle$ which produces a successor $\langle t, i-1\rangle$. Because t is a successor state of s, and we know that $s \not\models \mathcal{L}_{i-1}$, then $t \not\models \mathcal{L}_{i-2}$, as if it was, then s would not be forbidden in $\mathcal{L}_{i-1}$. As such, taking $j = i - 1$, when the obligation $\langle t, j\rangle$ is added to the queue, $t \models \mathcal{L}_j$ and $t \not\models \mathcal{L}_{j-1}$. ■

Lemma 5 ($\mathcal{L}_0$ is weaker than or equivalent to $\mathcal{G}$ constrained by the invariant clauses)

Proof. This is true initially (Line 1) and can only be violated when a new clause is added to $\mathcal{L}_0$ (Line 15). No produced state will ever violate the invariant clauses, by definition of the invariant clauses. If ever a state s was produced such that $s \models \mathcal{L}_0$, then by the invariant of Lemma 4 it would have to be at index zero, which would lead to PDR terminating having found a plan. As such, it could never form a reason constraining $\mathcal{L}_0$. ■

Theorem 1 (Given a planning problem $\langle X, A, I, G\rangle$ the algorithm terminates and returns *True* if and only if the problem has a plan.)

Proof. The algorithm returns True *in two instances (Lines 2 and 10). For Line 2, the initial state satisfies the goal condition so the problem is trivially solved. The queue initially holds just the initial state, and states are only added to the queue when they are successors of states on the queue, so all states on the queue are reachable from the initial state. For Line 10, a goal state is found which is a successor of a state in the queue, and as such is reachable from the initial state, so a plan exists.*

If PDR returns False *(Line 31) then the path construction phase (loop at Line 5) of an iteration of k (Loop at Line 3) has finished and there exists a j, $0 \leq j \leq k$ such that $\mathcal{L}_j = \mathcal{L}_{j+1}$. By combining the invariants in Lemmas 1, 2 and 5, and this equality, we have: $\mathcal{G} \models L_j$ and $L_{j+1} \wedge T \models L_j$. This combined with the invariant from*

Lemma 3 ($I \not\models \mathcal{L}_k$) shows that no plan can exist at any length.

We now prove termination. PDR always takes obligations from the queue with the lowest index. It follows from Lemma 4, that if the obligation has a successor it has never been seen before in the k-iteration (the successor state is currently the only state satisfying $\mathcal{L}_{i-1}$). Whenever a obligation is to be progressed, it might produce a successor, which can only happen a finite number of times, because each time a successor is generated it is unique and there are a finite number of states. Otherwise it will result in the addition of a new reason to the layer formula, which can also only happen a finite number of times. As such the "Q-processing" loop (starting line 5) will terminate. There are finitely many propositions and as such finitely many possible different layer formula. Because successive layer formulae can only be tighter, there is a maximum k that can be reached before two successive layer formulae are equivalent (and False *will be returned). Since the "Q-processing" loop and the clause pushing loop both terminate eventually, and there is a finite number of k-iterations, PDR will eventually terminate.* ■

2.5.4 Related Work

The first PDR algorithm was IC3, which stood for *"Incremental Construction of Inductive Clauses for Indubitable Correctness"* (Bradley 2011). The algorithm is a SAT-based procedure for verification of safety properties of transition systems without explicitly unrolling the transition relation, Property Directed Reachability was a phrase later coined in (Eén et al. 2011). IC3 featured in the 2010 Hardware Model Checking Competition (HWMCC'10), and a range of serial PDR procedures were subsequently adapted, developed, and evaluated for planning (Suda 2014). Of special interest here is PDRplan, a fast bespoke implementation of PDR that takes advantage of the common structure of many classical planning benchmarks—e.g., uniformly positive action preconditions—replacing all SAT inference with specialised constraint processing procedures that perform guaranteed polynomial time inference.

The question of how to use distributed computing to improve the runtime of PDR is of interest in our setting. We propose a novel parallel variety of PDR in Chapter 4. In the original IC3 manuscript (Bradley 2011), the author, Bradley, recognised the potential for accelerating PDR using distributed computing. He introduced a portfolio scheme, using a small set of IC3 processes that synchronise so that clause pushing (Alg. 2, Lines 26-29) can be performed by one serial process, and problem safety (Alg. 2, Line 31) can be detected by that process where applicable. In this original work, search diversity is enhanced using a non-deterministic SAT-solver

ZChaff (Vizel et al. 2015). IC3 portfolio members share reasons periodically via a central coordinating process and, when an IC3 instance receives reasons found by other processes, it removes all obligations from its queue that are inconsistent with those reasons. These two factors, the nondeterminism of the SAT-solver and the sharing of reasons is the backbone of the way the elements of the portfolio differ. With these modifications, Bradley was able to demonstrate that a portfolio using 12 cores can complete an additional 12 proofs in HWMCC'10 competition settings, compared with a serial IC3 baseline.

Preliminary investigations of portfolio PDR are extended in (Chaki and Karimi 2016), where a variety of strategies for distributed portfolios with reason sharing between members are described. Designed for hardware model checking problems, which can have many initial states, each portfolio member is a variant of IC3 that uses the deterministic SAT solver MiniSAT (Eén and Sörensson 2003). Portfolio members perform their own clause pushing, thus there is no need to synchronise the individual portfolio elements. In the two best strategies described, portfolio members are required to check if the portfolio has proved the problem *safe*. One strategy requires each portfolio element to derive their own proof, and another has portfolio elements consider the cumulative knowledge of all portfolio members, checking if the problem is *safe* when the process is due to increment k. The headline contribution is a detailed statistical and empirical analysis of portfolios in hardware benchmarks. In (Chaki and Karimi 2016), the exploration of portfolios in software verification is left as future work, and that challenge is taken up by Marescotti et al.(2017). Those authors develop a divide-and-conquer portfolio approach using Spacer (Komuravelli et al. 2016), "*partitioning*" the problem at hand syntactically into a set of subproblems, so that the concrete problem is *safe* iff every subproblem is *safe*. In addition to the innovation of partitioning, the authors also develop a "heuristic" for how members of the portfolio share reasons and search.

This history is of particular note to Chapters 4 and 5. In these chapters we adapt these existing parallel approaches into a single solver applicable to the planning setting. We then evaluate this against our own new parallel PDR solvers.

2.6 Decomposition

We here introduce some concepts related to decomposing planning problems – taking *concrete* problems and converting them into a collection of smaller problems by analyzing the problem's structure. This is used in the decompositional PDR approach described in Chapter 5.

We will find it convenient to refer to abstract planning problems and subproblem artefacts, defined by *restrictions* and *projections* to a subset $Y \subseteq X$ of the planning propositions. For a set of actions A and a set $Y \subseteq X$, the restriction $A{\downarrow}Y$ is the subset of actions in A that can be described completely if we restrict ourselves to using the symbols from Y only:

$$A{\downarrow}Y = \{a \in A | \Sigma(\mathit{pre}(a)) \cup \Sigma(\mathit{eff}(a)) \subseteq Y\}$$

In the case of a cube s and a set Y, the projection $s{\downharpoonright}Y$ is the cube projected to elements in Y. For example:

$$(\neg a \wedge \neg b \wedge c){\downharpoonright}\{a, c\} = (\neg a \wedge c)$$

We write $X^{\pm}$ to denote the set of elements in X that occur only positively or only negatively in the effects of actions. That is, for each element $x \in X^{\pm}$ only one of either x, or $\neg x$ appears in the conjunction $\bigwedge_{a \in A} \mathit{eff}(a)$. The problem *dependency graph* was first described by (Knoblock 1994; Williams and Nayak 1997). We present the slight variation that is motivated in our setting, based on (Rintanen and Gretton 2013). Although the concepts generalise, here we restrict our attention to STRIPS problems.

Definition 1 (Dependency Graph)
A dependency graph for a planning problem $\langle X, A, I, G\rangle$ is a directed graph (V, E) with vertices V in one-to-one correspondence with X, and an edge $(x, x') \in E$ for each pair of vertices where:

- *There exists an action a where $x, x' \in \Sigma(\mathrm{eff}(a))$ and $x' \notin X^{\pm}$. Note here if (v, v') is an edge (v', v) will be also. Or:*
- *There exists an action a such that $x \in \Sigma(\mathrm{eff}(a))$ and $x' \in \Sigma(\mathrm{pre}(a))$.* ■

For a graph (V, E) with $v, v' \in V$, we write $v \overset{(V,E)}{\rightsquigarrow} v'$ to signify that there is a path from v to v'. We also define the *Problem Specific Dependency Graph* (PSDG), a variant of the Dependency Graph which only includes variables relevant to achieving the goal.

Definition 2 (Problem Specific Dependency Graph)
Given a dependency graph (V, E) for problem $\langle X, A, I, G\rangle$, the problem specific dependency graph is obtained by removing all vertices (and thereby their adjacent edges) that are not in the set $\{v' | \exists v \in \Sigma(G), v \overset{(V,E)}{\rightsquigarrow} v'\}$. ■

This can be thought of as a general dependancy graph, without some of the propositions and actions which are known to not be useful when trying to reach a goal state.

Definition 3 (Strongly Connected Component (SCC))
A SCC of a directed graph is a subgraph in which there is a directed path from each

vertex to every other vertex, and every vertex v' mutually reachable from a vertex v in the subgraph is also in the subgraph – i.e., SCCs are maximal. ■

The SCCs of a graph can be computed in time linear with respect to the number of edges and vertices by *Tarjan's* algorithm, which is used in this thesis (Tarjan 1972).

Definition 4 (Lifted Acyclic Dependency Graph (LADG))
Given a dependency graph (V, E), a labelled directed acyclic graph $(\mathbf{P}, \mathbf{E})$ is a corresponding LADG iff:

- *Vertices $\mathbf{P}$ are bijectively labelled by members of a partition of variables V, and*
- $\mathbf{E}$ *contains an edge $(\mathbf{p}, \mathbf{p}')$ iff there is an edge $(v, v') \in E$ such that v appears in $\mathbf{p}$ and v' appears in $\mathbf{p}'$. This definition follows that of a lifted dependency graph described by Abdulaziz et al. (2018).* ■

A natural LADG has one vertex for each SCC in the dependency graph, with that set of SCCs providing a suitable partition. For example, taking one SCC as each member of the partition in the above definition, we arrive at a LADG. However, this is not the only LADG. In Chapter 5 we present an algorithm which uses the SCCs of a PSDG to create a LADG. However, further LADGs are created by merging vertices of the initial LADG, so the vertices no longer directly correspond to SCCs of the PSDG.

Example 6 (Analysing a Logistics Domain)
Consider the running Logistics example presented in Example 1/4. Figure 2.1 shows in solid arrows the dependency graph between propositions in the problem, and the dotted ovals and arrows show a corresponding LADG with vertices corresponding to SCCs.

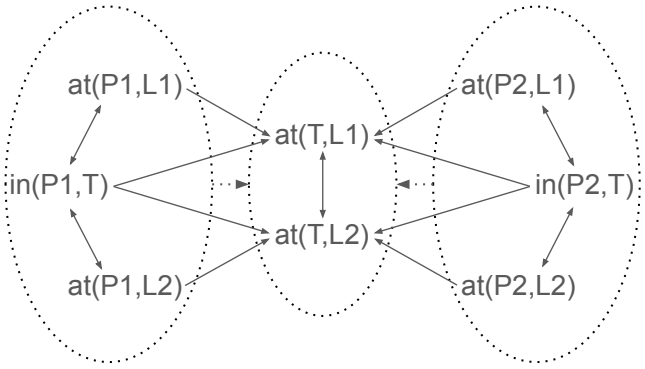


Figure 2.1: Logistics dependency graph, depicted with a LADG whose vertices, indicated by dashed ovals, correspond to SCCs of that dependency graph.

2.6.1 Related Work

In Chapter 5 we develop a PDR based approach based on the dependency graph concept that was first described in (Knoblock 1994; Williams and Nayak 1997). Intuitively, that idea is to synthesise a concrete plan through a process of iteratively refining plans using the abstraction hierarchy associated with the dependency graph. For example, in Logistics we first plan for the delivery of packages ignoring the detail of having to drive trucks, and then we attempt to refine that abstract plan, by interleaving the appropriate driving actions. This decompositional method is related to *decoupled search* Gnad et al. (2022). In decompositional search a star topology is used where a single center component interacts with many leaf components. As one point of contrast the method proposed in this thesis can take a wider variety of graphs, including arbitrary length chains and multiple root nodes. These seminal ideas were advanced in concert with literature related to (tree) decompositions (Darwiche 2001; Huang and Darwiche 2003; Robertson and Seymour 1991), with the planning literature developing conceptual frameworks and algorithms that fall under the banner of *factored* planning (Amir and Engelhardt 2003; Brafman and Domshlak 2006). These ideas are showcased and contrasted in relation to the dTreePlan planning system in (Kelareva et al. 2007). Unlike factoring/abstraction-refinement, our approach forms concrete plans by a simple concatenation operation, and not by sub-plan/action interleaving. Our approach is enabled because we "edit" the goals of abstract subproblems, thereby ensuring that—at least in practice on common planning benchmarks—our subplan concatenation operation yields a valid solution to the concrete problem at hand. In this last respect, our contribution is related to (Abdulaziz et al. 2015), a work in which the authors employ a goal editing idea to plan in systems composed of a set of symmetric subsystems.

Chapter 3

Multi-Step SAT in PDR

Modern SAT solvers are very powerful, designed for and capable of solving complicated problems. However, their use so far in PDR has been for solving a large collection of related small and easy SAT problems. The motivation for this chapter is to consider the benefits that might be obtained if the SAT calls were larger, offloading some of the processing from the PDR process to the highly effective SAT solvers. If taken to its extreme, this could effectively remove the PDR logic from the algorithm, reducing it back to a traditional SAT planning approach. We suppose there is a middle ground which can still utilize PDR, while also benefiting from SAT solver's ability to solve large problems effectively.

To do this, we propose processing multiple planning steps within a single SAT call. This involves having multiple corresponding *intermediate* states represented in the SAT instance. We propose two methods to do this, *PDR Macro* (PDR-M) and *PDR Interleaved Layers* (PDR-IL).

PDR-M involves allowing multiple planning steps between the state being progressed and the successor state. PDR-IL further constrains these intermediate time-steps to be themselves constrained by additional layer formulae.

3.1 PDR Macro

For our first variation PDR-M, instead of the SAT call corresponding to progressing a state satisfying layer $\mathcal{L}_i$ to a state satisfying $\mathcal{L}_{i-1}$ in a single step, we allow $\mathcal{F}$ steps to arrive at the successor state. More specifically, we replace $\mathcal{T} \wedge \mathcal{L}'_{i-1}$ on Lines 5 and 28 of Algorithm 2, with:

$$\left(\bigwedge_{n=0}^{n=\mathcal{F}-1} T^n \right) \wedge \mathcal{L}^{\mathcal{F}}_{i-1} \tag{3.1}$$

The successor state t is then extracted from time-step $\mathcal{F}$, which is the time-step con-

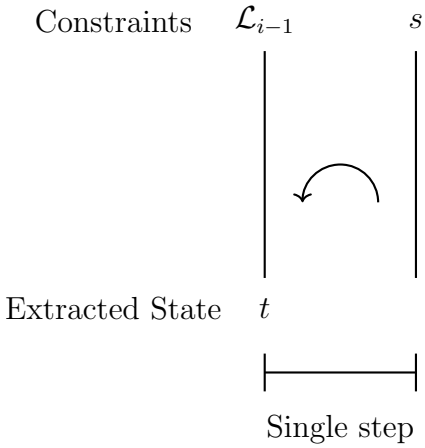


Figure 3.1: PDR's single step progression semantics. Vertical lines indicate time-steps, and the arc represents the transition.

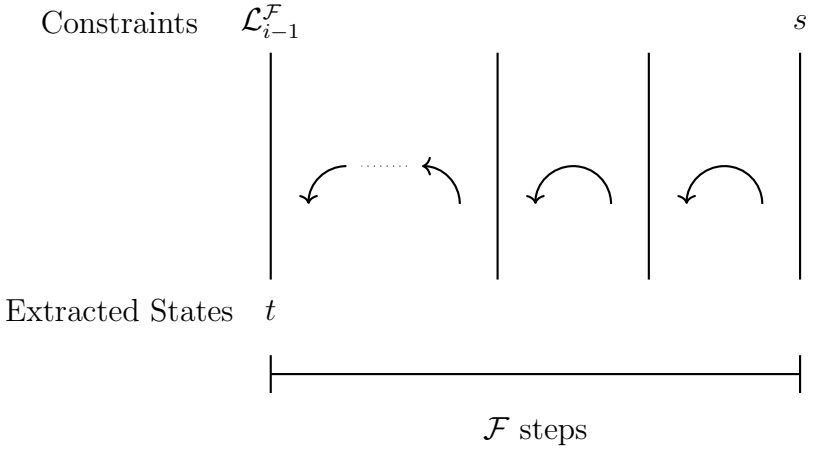


Figure 3.2: PDR-M multi step progression semantics. Vertical lines indicate time-steps, and arcs represent transitions.

strained by $\mathcal{L}^{\mathcal{F}}_{i-1}$. States found in the intermediate time-steps are ignored. Figures 3.1 and 3.2 graphically highlight the difference between the progression semantics of PDR and PDR-M.

Also note that when obligation rescheduling is not used, PDR will find the shortest plan[1], but PDR-M may not. By allowing multiple transitions in a single time-step, the effect of PDR-M can be simulated by modifying the original planning domain. For a set of actions in a planning problem A, an alternative set A' can be created where each action in A' is a sequential combination of actions in A. PDR-M then effectively operates with this alternate set of actions.

The first SAT call in PDR asks if the initial state can be transitioned to the goal

[1]Since in the $\forall$-step encoding multiple actions can be scheduled in the same time-step, a parallel plan with the least number of steps will be found. If trying to find a plan with the least number of actions, the progresison semantics must be changed so only one action can execute per time-step.

in one step. In PDR-M, this first call asks if the initial state can be transitioned to the goal in $\mathcal{F}$ steps, equivalent to a time-step bounded SAT planning call. much of PDR's efficiency comes from the how layers are progressively tighter closer to the goal, this guides PDR's choice of states to progressively step closer and closer towards the goal. Large $\mathcal{F}$'s can undermine this. In PDR-M, the first SAT call asks if the initial state can transition to a state satisfying $\mathcal{L}_0$. When it cannot, $\mathcal{L}_1$ is constrained solely by a reason derived from the initial state and the next SAT call asks if the initial state can transition to a state satisfying $\mathcal{L}_1$. With large enough $\mathcal{F}$'s this SAT call effectively samples arbitrary reachable states, undermining the guided/progressive nature of the layer formulae.

3.2 PDR Interleaved Layers

We propose another multi-step transition variation PDR-IL which constrains the intermediate time-steps with the layer formulae. The successor time-step is constrained by $\mathcal{L}_{i-1}$ as in standard PDR, but further successor time-steps are constrained by further layer formulae. More specifically we replace $\mathcal{T} \wedge \mathcal{L}'_{i-1}$ on Lines 5 and 28 of Algorithm 2, with:

$$\bigwedge_{n=0}^{n=\mathcal{B}-1} (\mathcal{T}^n \wedge \mathcal{L}_{i-n-1}^{i+n+1}) \tag{3.2}$$

Where $\mathcal{B} = min(\mathcal{F}, i)$. $\mathcal{B}$ plays a similar role in Eq 3.2 as $\mathcal{F}$ does in Eq 3.1, but allows for the case where i is too small, and there are not enough layers. Unlike PDR-M, because the intermediate states are constrained by the layer formulae, corresponding obligations can be extracted and added to the queue. More specifically, each successor state t^n at each successor time-step $1 \leq n \leq \mathcal{B}$, the obligation $\langle t^n, i-n \rangle$ is added to the queue. Figures 3.1 and 3.3 graphically highlight the difference between the progression semantics of PDR and PDR-IL.

In order to declare that no plan exists, unlike regular PDR where two subsequent layers have to be identical, here $\mathcal{F}+1$ layers have to be identical. To illustrate, consider the case when $\mathcal{F} = 2$, and any plan requires more than two non-parallelizable actions. Also, reason minimization is disabled, so when an obligation fails to be progressed, the whole state is used as the reason.[2] In this case, when $k = 1$, an attempt is made to progress $\langle I, 1 \rangle$, which fails setting $\mathcal{L}_1 \equiv \neg I$. k will then be incremented, and similarly, $\langle I, 2 \rangle$ will also fail to progress, setting $\mathcal{L}_2 \equiv \neg I$. At this

[2]We add this for ease of explaining this example, but there exists problems where no reason minimization is possible, and all reasons will be full states.

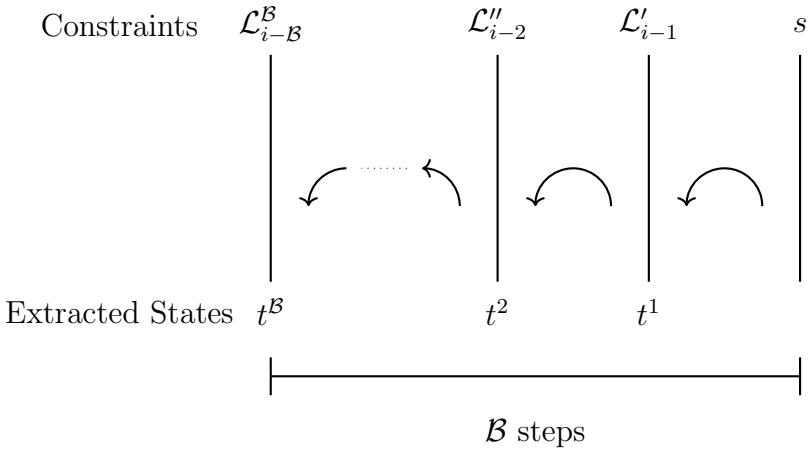


Figure 3.3: PDR-IL multi step progression semantics. Vertical lines indicate time-steps, and arcs represent transitions.

point, $\mathcal{L}_2 \equiv \mathcal{L}_1$, triggering the unsatisfiable stopping condition, but the problem may yet be solvable.

Like PDR-M, PDR-IL has a side effect when interacting with obligation rescheduling. When obligation rescheduling is disabled, then for each obligation $\langle t, i\rangle$ in the queue there is also a parent obligation $\langle s, i+1\rangle$ in the queue where there is a one step transition from s to t (unless t is the initial state). When obligation rescheduling is enabled, this does not hold, an obligation $\langle s, i+1\rangle$ can have a child obligation $\langle t, i\rangle$, which can then be rescheduled to $\langle t, i+1\rangle$. This process can be repeated, leading to long chains of successive obligations all at the same layer. When a $\mathcal{B} > 1$ PDR-IL procedure fails to progress an obligation $\langle s, i\rangle$, this conveys that there is no sequence of steps from s where each successive state satisfies a successive layer formula from $\mathcal{L}_i$. This is a tighter constraint than what a layer represents when obligation rescheduling is enabled. It is however consistent with the layer semantics when obligation rescheduling is disabled.

In the extreme case where $\mathcal{F} = \infty$, this will revert to an approach similar to a linear $1, 2, 3...$ step ramp-up query strategy. At each k-iteration, there will be a single SAT call, asking if the initial state can transition to the goal, with each intermediate step i bound by the sole reason derived from the initial state being unprogressable when $k = i$.

Note that when $\mathcal{F} = 1$, PDR-IL and PDR-M are equivalent to each other and to traditional PDR, as only a single transition is allowed.

3.3 Experimental Evalutation

The runtime performance of PDR-M and PDR-IL compared to baseline PDR is evaluated. All experiments here were performed on an Intel Xeon Gold 6134 CPU@3.2GHz with 20GB Memory and a 2-hour (7200s) timeout. We implemented PDR with obligation rescheduling using *Lingeling* as the SAT solver (Biere 2017). Invariants were generated from the Plangraph as specified in (Robinson et al. 2009). The runtime performance of all PDR planners is dominated by the time spent in the underlying SAT procedure, which is what is measured and reported in this section.

Our implementations were tested on a subset of benchmark planning problems from the International Planning Competitions up to 2014. Recall that when $\mathcal{F} = 1$, PDR-M, PDR-IL and traditional PDR are equivalent. We refer to this solver as the *baseline*.

We observed that the runtime performance of planners could vary significantly for different values of $\mathcal{F}$. For example, the SAT times for different values of $\mathcal{F}$, for the problem *Thoughtful-bootstrap-typed-03* from IPC2014 using PDR-M are given in Table 3.1.

$\mathcal{F}$	1	2	3	4	5	6	7
Time (s)	1.3	1.0	0.6	0.8	0.7	6197.2	831.3

Table 3.1: SAT Times for *Thoughtful-bootstrap-typed-03* solved with PDR-M, using macro lengths 1 to 7.

If any planner in this comparison cannot solve a problem, for any value of $\mathcal{F}$, then we exclude that problem from reporting.

We have run each PDR procedure under investigation on each problem, using values of $\mathcal{F} \in \{1, .., 7\}$. For each run we measured the total time spent by the planner in the underlying SAT procedure – i.e. here that is LINGELING. To report our experimental findings here, we have summarised the collected runtime data by reporting the average slowdown/speedup factor relative to the baseline, plotted in Figure 3.6. This is the average ratio of time spent relative to the baseline—i.e. a higher value means the planner with its $\mathcal{F}$ value was slower than the baseline. The runtime factor of the baseline procedure—i.e. taking $\mathcal{F} = 1$—is uniformly 1, and the runtime factors plotted for the other planners are the average values across the domain relative to this baseline. An example of this calculation is shown in the tables in Figure 3.4. The standard deviations between slowdown/speedup factors are shown in Table 3.2.

Problem	**Runtime (s)**		
	$\mathcal{F}$**=1**	**2**	**3**
Problem1	2	4	6
Problem2	3	9	12

Problem /Average	**Slowdown/Speedup Factor**		
	$\mathcal{F}$**=1**	**2**	**3**
Problem1	1	2	3
Problem2	1	3	4
Domain Average	1	2.5	3.5

Figure 3.4: Data processing example for a fictitious domain, for the results presented in Figure 3.6.

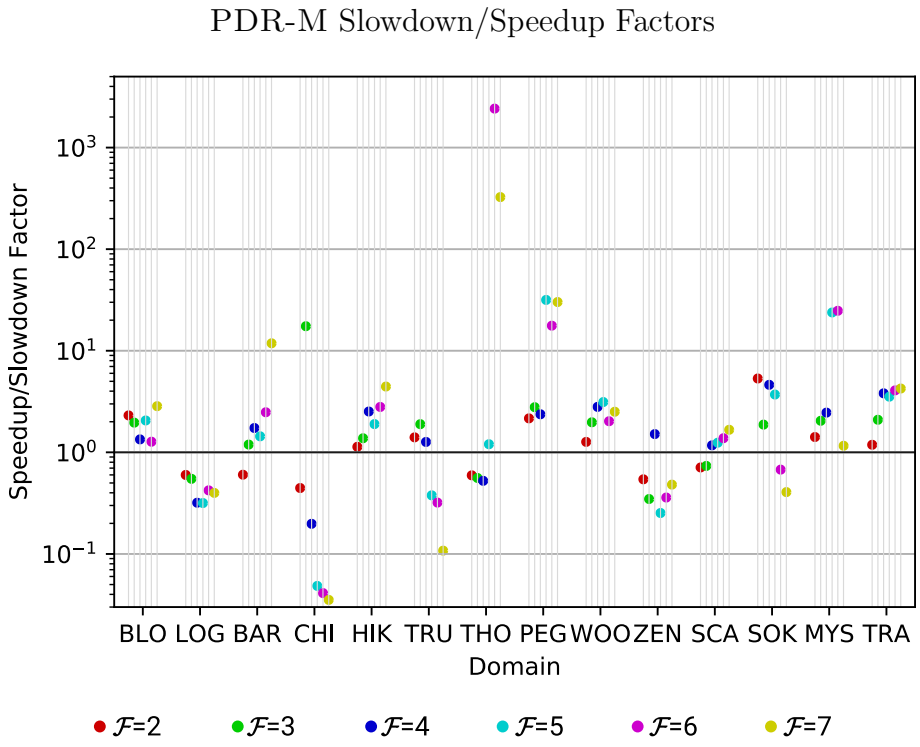


Figure 3.5: PDR-M average slowdown/speedup factors for each length $\mathcal{F}$ in $\{2, .., 7\}$. Vertical lines left to right indicate values for $\mathcal{F}$, 2 through 7 (Lower values indicate the variation is faster). Domain abbreviations are shown in Table 3.2.

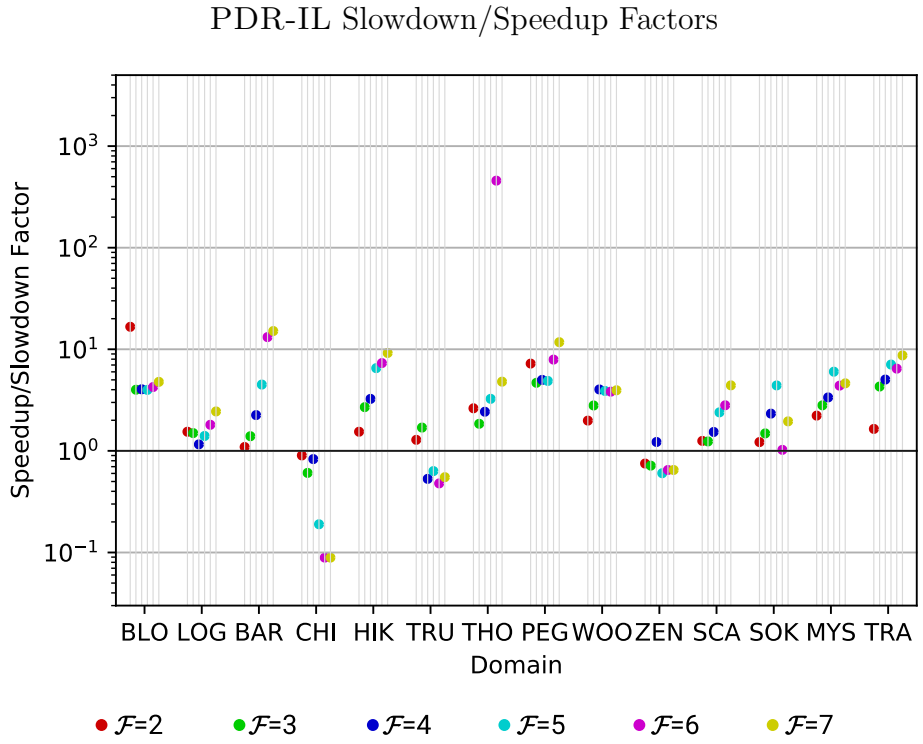


Figure 3.6: PDR-IL average slowdown/speedup factors for each $\mathcal{F}$ in $\{2, .., 7\}$. Vertical lines left to right indicate values for $\mathcal{F}$, 2 through 7 (Lower values indicate the variation is faster). Domain abbreviations are shown in Table 3.2.

Domain	PDR-M						PDR-IL					
(Abbreviation) (#Problems)	$\mathcal{F}$**=2**	**3**	**4**	**5**	**6**	**7**	**2**	**3**	**4**	**5**	**6**	**7**
Blocksworld (BLO) (88)	4.85	3.98	**0.99**	3.87	1.46	13.19	82.51	3.29	3.28	**2.77**	3.1	3.30
Logistics (LOG) (74)	0.34	0.58	0.22	**0.21**	0.24	0.23	**0.67**	0.81	0.74	0.84	2.4	3.91
Barman (BAR) (14)	**0.39**	1.09	2.78	1.31	2.3	13.52	**0.88**	0.96	1.71	4.21	29.14	20.87
Childsnack (CHI) (4)	0.45	34.50	0.36	0.05	0.05	**0.04**	1.46	0.71	1.53	0.22	**0.09**	**0.09**
Hiking Ptesting (HIK) (9)	0.53	**0.52**	1.93	1.02	2.76	2.44	**0.46**	1.13	1	5.02	5.58	6.16
Trucks (TRU) (2)	0.71	0.31	0.9	0.24	0.26	**0.02**	0.78	0.56	0.21	0.19	**0.02**	0.18
Thoughtful Bootstrap (THO) (2)	0.24	**0.11**	0.17	0.97	3,422.9	457.56	1.97	**0.2**	0.34	0.64	642.35	0.37
Pegsol (PEG) (9)	**1.38**	2.23	1.99	50.25	34.22	72.78	13.24	5.31	4.37	**3.67**	4.58	9.31
Woodworking (WOO) (20)	**0.42**	0.78	1.33	1.75	0.95	1.18	**0.55**	1.03	1.64	1.63	1.59	1.63
Zenotravel (ZEN) (6)	0.54	**0.26**	1.15	**0.26**	0.35	0.58	0.82	0.7	1.12	0.52	0.55	**0.5**
Scanalyzer (SCA) (3)	0.41	**0.33**	0.67	0.6	0.92	1.01	1.14	**1.02**	1.50	2.82	3.53	6.34
Sokoban (SOK) (8)	8.73	2.57	11.61	9.54	0.94	**0.53**	1.94	2.1	2.59	9.83	**1.05**	3.08
Mystery (MYS) (7)	1.51	1.65	2.11	58.94	61.67	**0.86**	**1.99**	2.48	2.71	7.59	4.06	4.29
Transport (TRA) (5)	**0.49**	1.15	2.31	1.87	3.85	5.47	**0.79**	4.01	2.37	5.83	4.50	10.26
Num. Domains with Lowest stddev.	**4**	**4**	1	2	0	**4**	**6**	2	0	2	3	2

Table 3.2: Standard deviation of slowdown/speedup factors for PDR-M and PDR-IL. The bold result is the lowest standard deviation for each solver for each domain. The domains are identified by: their full name (abbreviation) (number of problem instances solved). In order to report a speedup factor, we can only consider instances that all systems can solve.

3.4 Conclusion

We here introduced two PDR based solvers which augment the obligation progression SAT call in PDR, by allowing multiple time-steps as opposed to just a single one. In general, we find the impact of this change to be highly varied, and very dependant on the domain chosen. PDR-M allows multiple steps from the originating state to the successor state, PDR-IL however keeps a single step to the first successor state, but allows multiple further successor states constrained by further layer formula. It is more common to be able to speedup planning using PDR-M than when using PDR-IL. Notably in the Logistics domain, for PDR-M there is a consistent speedup for all $\mathcal{F}$ values tested, especially for $\mathcal{F} = 4$, with a low standard deviation of 0.22.

For both PDR-M and PDR-IL we find that lower values of $\mathcal{F}$ correspond to lower standard deviations in the slowdown/speedup factors. Similarly, higher $\mathcal{F}$ values tend to correspond to slowdown/speedup factors further away from one – i.e. increasing $\mathcal{F}$ tends to increase the variance in performance compared to the baseline.

For PDR-M, the results of the multi-step SAT call could be replaced by those of a single step SAT call, over a different problem formulation in which the actions are replaced by a chained or sequenced set of these actions, with appropriate limits on the number of actions. In this way, PDR-M can be thought of as operating as normal PDR with a different presentation of the same set of actions. Because of this, we expect and experimentally find the performance of PDR-M to be highly dependant on the domain. A less direct parallel cannot be drawn of PDR-IL, but for similar reasons we also expect it to be highly dependant on the domain.

Chapter 4

Parallelizing Obligation Processing in PDR

Since its inception, the potential for parallelism in PDR has been identified (Bradley 2011). Classical PDR iteratively takes a single state from the queue of work to be done, and processes it before continuing. We here develop *Parallel State PDR* (PS-PDR), a sound and complete variant of PDR which maintains a collection of compute nodes all working in parallel. This collection of nodes is used to process states independently in parallel.

There exists parallelisms of PDR (Bradley 2011; Chaki and Karimi 2016; Marescotti et al. 2017). However these are not suitable for planning problems and so cannot be directly compared to our incumbent parallel approach. Because of this, along with PS-PDR we also create a solver designed to adapt these existing techniques to the planning setting, and evaluate it against our approach.

In Chapter 5 we create another parallel PDR solver, that can be run on the same benchmarks, and uses the same codebase as PS-PDR. Because of this, we delay our experimental evaluation of PS-PDR to instead be included in Chapter 5, so PS-PDR can be compared against this other solver as well.

4.1 Overview and Preamble

PS-PDR is a sound and complete planning procedure described given an allocation of n computing cores, which in our context is in a High Performance Computing (HPC) environment. One core is designated the *orchestrator* and the remaining $M = n - 1$ cores are designated *workers*. Each worker is given a unique worker ID from $\{1, .., M\}$. The orchestrator conducts the bulk of the "PDR logic" while the workers process the obligations. PS-PDR has many similarities to classical PDR, including that when $M = 1$, PS-PDR simulates classical PDR. When $M > 1$ the following provides the control flow regarding how the PS-PDR orchestrator

coordinates with workers to implement a distributed PDR algorithm.

Similar to PDR, PS-PDR maintains layer information about the system, and proceeds by taking an obligation $\langle s, i \rangle$ from the queue, and querying whether or not there exists a successor state t which satisfies the layer formula $\mathcal{L}_{i-1}$. If a successor state exists a corresponding obligation is produced and added to the queue, otherwise a reason is produced and used to constrain the layer formula.

Algorithm 3 provides an exposition of PS-PDR. There, a variety of functions are introduced to facilitate the operation of the PS-PDR orchestrator which we will describe here:

- The *workers_waiting* function (Lines 5, 6, 27 and 28) provides the set of workers which are not currently processing obligations.
- The *process_obligation* function (Line 9) encapsulates an assignment to a worker, which will be to process an obligation $\langle s, i \rangle$ in the context of the orchestrator's Layer information. Our pseudocode indicates an entire layer formula is communicated from the orchestrator to a worker at the time of a work assignment. For efficiency, our implementation communicates only the clauses added since the last assignment to the indicated worker. If a valid successor state t from the obligation $\langle s, i \rangle$ is found by a worker, the tuple $\langle t, s, i \rangle$ is sent from the worker to the orchestrator and stored in a buffer. A valid successor may not exist, in which case the worker derives a reason r by iteratively calling an incremental SAT procedure as per standard PDR. Here, a tuple $\langle r, i \rangle$ is sent to the orchestrator and stored in a separate reasons buffer.
- The *get_successes* function (Line 10) retrieves then empties the buffer of successor tuples created by *process_obligation* since its last call.
- The $get\_failures$ function (Line 13 and 32) behaves as *get_successes* does, but with respect to the reasons tuples created by *process_obligation*.
- The *process_pushing* function (Line 31) behaves identically to *process_obligation*, except that instead of iteratively finding a minimised reason, the state s from the obligation is returned directly as the reason.

4.2 The Core Algorithm

First, the queue and layer information are initialised (Line 1). A check is made to see if the initial state satisfies the goal condition, and if so then the problem is trivially satisfiable (Line 2). After that, a loop over the length of planning horizons $k \in [1, 2, 3, ...]$ commences (Line 3). When the algorithm comes to increment k

Algorithm 3: Parallel State PDR

Input: A planning problem $\langle X, A, I, G\rangle$. The number of available cores M.
Output: *True* if the problem is solvable, *False* otherwise

```
1  Q ← {}, L_0 ← G, for j > 0 : L_j ← True
2  if I ⊨ L_0 then return True
3  for k ∈ [1, 2, 3, ...] do
4      Q ← {⟨I, k⟩}
5      while Q ≠ {}, or workers_waiting() ≠ {1, .., M} do
6          foreach w ∈ workers_waiting() do
7              if Q ≠ {} then
8                  ⟨s, i⟩ ← pop most recently added ⟨s, i⟩ from Q with minimal i.
9                  process_obligation(w, L, ⟨s, i⟩)
10         foreach ⟨t, s, i⟩ ∈ get_successes() do
11             if i − 1 = 0 then return True
12             Q ← Q ∪ {⟨s, i⟩, ⟨t, i − 1⟩}
13         foreach ⟨r, i⟩ ∈ get_failures() do
14             for j ∈ {0, .., i} do
15                 L_j ← L_j ∧ ¬r
16             if i < k then Q ← Q ∪ {⟨s, i + 1⟩}
17             // Trim the queue with the new reason
18             Q_trimmed ← {}
19             foreach ⟨z, h⟩ ∈ Q do
20                 if h > i or UNSAT(r ∧ z) then
21                     Q_trimmed ← Q ∪ {⟨z, h⟩} // Keep as is
22                 else if i < k then
23                     Q_trimmed ← Q ∪ {⟨z, i + 1⟩} // Keep at a higher layer index
24             Q ← Q_trimmed
25     for i ∈ {1, .., k + 1} do /* Clause Pushing */
26         CP ← {⟨¬c, i⟩ |c ∈ L_{i−1}, c ∉ L_i}
27         while CP ≠ {} or workers_waiting() ≠ {1, .., M} do
28             foreach w ∈ workers_waiting() do
29                 if CP ≠ {} then
30                     ⟨¬c, i⟩ ← pop from CP
31                     process_pushing(w, L, ⟨¬c, i⟩)
32             foreach ⟨¬c, _⟩ ∈ get_failures() do
33                 L_i ← L_i ∧ c
34         if L_{i−1} ≡ L_i then return False
```

without having found a plan, then $\texttt{UNSAT}(\texttt{I} \wedge \mathcal{L}_\texttt{k})$, and therefore there is no plan with k or fewer steps. At the beginning of iteration k the obligation $\langle I, k\rangle$ is pushed to Q,

and then a loop starting on Line 5 is entered with the break condition: $Q = \{\}$ and $workers\_waiting() = \{1, .., M\}$. The first condition, $Q = \{\}$, is familiar to PDR, indicating that work at k continues until there are no obligations in Q to evaluate. The second condition, $workers\_waiting() = \{1, .., M\}$, is peculiar to PS-PDR, and means the orchestrator cannot increment k until all workers have completed their evaluations of obligations, and therefore buffered any further obligations to Q as required.

Evaluated obligations that are satisfiable are retrieved and processed (Lines 10-12). Similarly to classical PDR, for the obligations that are progressable, an obligation for the successor state along with the original obligation are added to the queue. For the obligations that are not progressable, a corresponding reason is generated and used to constrain the layer information, and the original obligation is added back to the queue at a higher index (Lines 13-15). As in classical PDR, the queue is then trimmed to be consistent with the layer formulae (Lines 18-24). Once the queue is empty and all workers are free, identically to classical PDR, clause pushing is performed and a test for termination is evaluated (lines 25-34).

4.3 Analysis

We implement and evaluate PD-PDR. However, the solver presented in Chapter 5 is built on the same software framework, and is evaluated on the same benchmark set as PS-PDR. Additionally, it is also a parallel version of PDR. Because of this, the experimental evaluation of PS-PDR is provided alongside that of this other solver in Section 5.4, to allow these solvers to be compared against each other.

4.3.1 Comparison to, and Adaption of Existing Parallelizations

Section 2.5.4 presents and reviews existing parallelizations to PDR in the literature. PS-PDR is unique to these incumbent solvers for three primary reasons:

- These solvers all employ a portfolio approach, having a set of serial PDR processes running in tandem, sharing learnt layer clauses. This is distinct from our approach, where a central orchestrator maintains a central queue and farms work out to external processes.
- These solvers are not designed for classical planning, and cannot parse STRIPS PDDL.

- They do not perform obligation rescheduling, which we find to be very important for performance.

However, we try to compare the core ideas of PS-PDR to the ideas presented by these other solvers. We create and evaluate an additional solver we call *PDR-Portfolio* (PDR-P), which we designed to try capture the "collective essence" of these existing methods, with the addition of obligation rescheduling. PDR-P behaves similarly to PS-PDR, except that instead of having a sole queue for the whole solver, the orchestrator maintains a separate queue for each worker. When workers request/return obligations, these are retrieved from/added to their respective queues. This also has the effect of having a global k-value, so if one queue is empty, no work is done by the corresponding worker until the other queues are also empty. Additionally instead of pushing layer clauses independently, this work is distributed amongst the workers as their layer information is synchronized. This is a mechanism observed in existing parallelizations of PDR. Here, the sharing of clauses, and minor asynchronicities creates nondeterminism within the solver's searches. Nondeterminism is created by the sharing of clauses, causing the workers/queues to fall out of sync. This solver is based on the same codebase as PS-PDR, which helps ablate implementation differences, to perform a "like for like" comparison with PS-PDR.

4.3.2 Performance Limits with Increased Cores

We find that as the number of cores is increased, the performance of PS-PDR tapers off. When processing obligations in parallel, work done can be repeated/unnecesary. To demonstrate this, consider two examples:

1. Consider the case where there are many similar obligations on the queue with the same layer index. They are all unprogressable, and processing them will produce the same reason. In serial PDR, one will be taken off the queue, processed, then the reason produced would trim the rest. Whereas in PS-PDR, all such obligations could be taken off at the same time. All processes would return the same reason, wasting the workers time.
2. Consider a domain with a set of states $\{s_0, s_1, s_2, ..., s_x\}$, where $s_0 \equiv I$ and $s_x \equiv G$, and the only actions available transition s_z to s_{z+1} for $z < x$. The queue PDR/PS-PDR maintains would contain the set of states $\{s_0, s_1, s_2, ..., s_y\}$. When processing any corresponding obligation, only processing the obligation corresponding to s_y would be able to generate a successor state that had not been encountered before. As such, processing multiple obligations in parallel would offer no benefit over processing them serially.

4.4 Conclusion

We here present a parallel variation of PDR which processes multiple obligations independently in parallel. This is in contrast to existing parallelizations of PDR in non-planning contexts which maintain a portfolio of PDR solvers sharing knowledge amongst themselves through various synchronization mechanisms.

Remarks on the experimental evaluation are saved for Chapter 5.

Chapter 5

Parallel Decompositional PDR

We here again use distributed computing environments to accelerate PDR through parallelism. However, instead of processing obligations in parallel as PS-PDR does, we here decompose the problem into a series of *subproblems* by analyzing its dependency graph. We then solve these subproblems independently in parallel, and attempt to concatenate the resulting plans into a plan for the original *concrete* problem. If this fails, subproblems are combined and the process repeats. In the worst case, subproblems are combined to effectively recreate the concrete problem, which is solved directly. We call the resulting sound and complete solver *Parallel Decompositional PDR* (PD-PDR).

To help the exposition of PD-PDR, we first give an overview of component concepts in Section 5.1, before defining them more formally in Section 5.2. We then provide Examples 7 and 8, showing the execution of PD-PDR.

5.1 Algorithm Overview

We here provide an overview of PD-PDR, with less precision than the formal description that we will provide below in Section 5.2. This section should also serve as an informal glossary of relevant terms.

We present an intuitive scenario to demonstrate the core idea of the operation of PD-PDR through analogy. Consider a communal desk, which many people need to use to process their papers. The desk is initially very ordered and tidy, with a stack of unprocessed papers, a stack of processed papers, and a one hour pass to a laminating machine. One by one, each person can go to the desk, take their unprocessed papers, and go about processing them. In the course of doing so, they can strew papers all over the desk, use a portable computer peripheries on the desk, et cetera. By the time they finish, they return everything back almost to how it was found, except that the processed papers are now in their relevant pile, and the laminating machine pass might have been used. The next person can then

come along and do the same thing, with no need to concern themself with how the previous person did their work, or any mess that used to be on the desk. As long as the laminating machine pass has not been used should they need it, then this next person can do their work.

In this example, the individual people represent the subproblems. The planning problem associated with each subproblem achieves a portion of the goal (some processed papers), while returning the rest of the environment to close to how it was found (clean, maybe without the laminating machine pass). Additionally, it may be the case that multiple people need to use the laminating machine pass to complete their task. All subproblems/people which may need it have access, and this only causes an issue if more than one subproblem requires use of it. If multiple subproblems require it, then a new iteration of new subproblems is created where the subproblems are merged – equivalent to coordinating so the two or more people who need the laminating machine pass come in and use it at the same time.

5.1.1 Terminology

The exposition of PD-PDR relies heavily on the background material presented in Section 2.6. As a summary, we review the following concepts:

- $\Sigma(F)$: For a formula F, $\Sigma(F)$ is the set of all propositions mentioned in F.
- PSDG: The dependancy graph is a graph of problem propositions, with an edge from v_1 to v_2 if the value of v_1 depends on v_2. The problem specific dependency graph (PSDG), is a variant of the dependency graph which only includes variables relevant to achieving the goal.
- LADG: A lifted acyclic dependency graph (LADG) is similar to the PSDG, but takes a more abstracted view. It is a graph (V, E) which is created from a corresponding PSDG. Each vertex in V is a set of propositions corresponding to an SCC of the PSDG. There is an arc v_1, v_2 iff there is a corresponding arc in the PSDG between a proposition in v_1 to a proposition in v_2.
- $X^{\pm}$: For a planning problem $\langle X, A, I, G\rangle$, we write $X^{\pm}$ to denote the set of elements in X that occur only positively or only negatively in the effects of actions.
- $A{\downarrow}Y$: For a set of actions A and a set $Y \subseteq X$, $A{\downarrow}Y$ is the subset of A that can be described completely if we restrict ourselves to using the symbols from Y only:

$$A{\downarrow}Y = \{a \in A | \Sigma(\mathit{pre}(a)) \cup \Sigma(\mathit{eff}(a)) \subseteq Y\}$$

- $s\lfloor Y$: For a partial or full state s and a set $Y \subseteq X$, $s\lfloor Y$ is the state projected to elements in Y. For example:
$$(\neg a \wedge \neg b \wedge c)\lfloor\{a, c\} = (\neg a \wedge c)$$

The algorithm proceeds in a series of *iterations*. If at the end of any iteration a concrete plan is found, that is returned and the algorithm stops. In practice, many problems can be solved with a single iteration. Each iteration is started with a LADG Γ. For the first iteration, it is created directly from the planning problem. In subsequent iterations, it is created from the result of the last planning problem. A series of properties are derived in sequence from Γ in each iteration:

- Δ_G is a list identifying subproblems for the current iteration, where each element is a vertex from Γ (each being a set of propositions). A vertex from Γ is contained in Δ_G iff that vertex contains a proposition mentioned in the goal condition. The ordering of Δ_G is constrained to be consistent with the partial order imposed by Γ. Each element of Δ_G represents a set of interlinked propositions, where some subsets need to be set a certain way to satisfy the goal condition. For example, in the running Logistics problem presented in Example 1/4, each element of Δ_G will be a set of propositions all describing the location of a specific package/truck.

- For each $\mathbf{p}_G \in \Delta_G$, $F(\mathbf{p}_G)$ is all of the propositions required to plan for $\mathbf{p}_G$. It is all the propositions in $\mathbf{p}_G$, as well as all the other elements of Γ $\mathbf{p}_G$ is dependant on.

- For each $\mathbf{p}_G \in \Delta_G$, $Dep(\mathbf{p}_G)$ is a set of propositions that need to be returned to their initial polarity in the final state reached by a plan for that subproblem. In the example above, this would correspond to the desk being tidy, and the other papers being left where they were found.

- For each $\mathbf{p}_G \in \Delta_G$, $Ex(\mathbf{p}_G)$ are the *exclusions*; a set of $\mathbf{p}_G$ related propositions whose truth value can only change once in a plan. Because they cannot change more than once in a plan, enforcing that they return to their original polarity in the subproblem goal enforces they cannot flip polarity at all. However, it may be necessary to flip their polarity to create a valid plan. We allow subproblems to flip the polarity of these variables and do not enforce they are returned back to their original polarity. If multiple subproblem plans flip the polarity of one of these variables, then the subproblems need to be reformulated for a subsequent iteration.

  In the above analogy of multiple users of the desk as subproblems, the single use laminator pass would be mentioned in the exclusions. If only one person

needs the laminator, they can just use it. However if multiple people need it, then they need to coordinate to use it together – equivalent to reformulating subproblems.

We note that while any sound and complete solver can be used here, we use PDR as it allows for a "like-for-like" comparisons with other PDR based solvers.

5.2 The Core Algorithm

Given a concrete planning problem $\langle X, A, I, G\rangle$ and its corresponding PSDG (V, E), PD-PDR proceeds by iteratively planning according to a sequence of increasingly contracted LADGs. At any iteration a concrete plan might be found, or otherwise the algorithm may prove that no plan exists. In each iteration a list, Δ_G, of subproblems is defined and then solved, with these determined efficiently according to a current LADG. The listed position of a subproblem's corresponding $\mathbf{p}_G$ in Δ_G identifies what goals must be achieved, and also the initial conditions that are obligated to be maintained by a valid plan for that subproblem.

5.2.1 Preliminaries

Listed subproblems are solved independently in parallel using PDR. A PDR process either produces a subproblem plan, or otherwise determines subproblem unsatisfiability. If every subproblem is solved, then a candidate concrete plan corresponds to the concatenation of subproblem plans. The algorithm terminates if that candidate is validated, or if the unsatisfiability of a subproblem indicates the concrete problem has no solution. If the algorithm does not terminate at an iteration, then some vertices of the LADG are contracted, and a subsequent iteration begins using the contracted graph. This iterative process eventually yields an LADG with a single vertex, at which point the algorithm behaves as PDR. That is, the only subproblem corresponds to the concrete problem at hand. Frequently in practice a valid plan can be found quickly within one or two iterations.

5.2.2 Construction of Subproblems

Through the course of its operation, PD-PDR may have many iterations. We here develop the details of an iteration of PD-PDR formally. Each iteration has a corresponding LADG Γ. For the first iteration, $\Gamma = (\mathbf{P}, \mathbf{E})$, with one vertex for each SCC in the PSDG (V, E). The details of finding the Γ for later iterations is described below. The subproblems posed at an iteration are defined as follows.

- Being directed and acyclic, Γ induces a natural partial order over $\mathbf{P}$. Given Γ, PD-PDR begins an iteration by creating a list, Δ_G, comprised of all the elements in $\mathbf{P}$ labelled with a goal proposition. The only ordering constraint is that a path from $\mathbf{p}_1$ to $\mathbf{p}_2$ in Γ constrains $\mathbf{p}_1$ to occur before $\mathbf{p}_2$ in Δ_G, and otherwise the order can be taken as arbitrary.

- For $\mathbf{p}_i \in \Gamma$, let $V(\mathbf{p}_i)$ be the propositions labelling $\mathbf{p}_i$.

- Each $\mathbf{p}_G$ in Δ_G is associated with a set of propositions $F(\mathbf{p}_G)$:

$$F(\mathbf{p}_G) \equiv \{f | \mathbf{p}' \in \mathbf{P}, \mathbf{p}_G \overset{\Gamma}{\rightsquigarrow} \mathbf{p}', f \in V(\mathbf{p}')\} \cup V(\mathbf{p}_G)$$

  $F(\mathbf{p}_G)$ contains the propositions that are required to plan for the goals in $\mathbf{p}_G$.

- Each $\mathbf{p}_G$ in Δ_G also has a set of exclusions, $Ex(\mathbf{p}_G)$. These are $\mathbf{p}_G$ related propositions whose truth value can only change once in a plan.

$$\begin{aligned} Ex(\mathbf{p}_G) \equiv \; & \{x | x \in X^{\pm}, v \in F(\mathbf{p}_G), a \in A, \\ & x \in \Sigma(\mathit{eff}(a)), v \in \Sigma(\mathit{eff}(a))\} \end{aligned}$$

- We assume we are provided with a binary mutex relation, identifying pairs of propositions that cannot be simultaneously *True* in a problem state. Let $\overline{M}(I, \mathbf{p})$ be all *True* propositions from the initial state I that are not in a binary mutex relation with propositions in $G \downarrow V(\mathbf{p})$. In words, the set of initial state propositions that can be *True* in a state where the goal, projected to propositions in $V(\mathbf{p})$, is satisfied.

  Let $i(\mathbf{p}_G)$ be the integer index of where entry $\mathbf{p}_G$ occurs in list Δ_G. The dependants $\mathit{Dep}(\mathbf{p}_G)$ are:

$$\begin{aligned} \mathit{Dep}(\mathbf{p}_G) \equiv \; & (\{f | \mathbf{p}' \in \Delta_G, i(\mathbf{p}') > i(\mathbf{p}_G), f \in F(\mathbf{p}')\} \\ & \cap F(\mathbf{p}_G) \cap \overline{M}(I, \mathbf{p}_G)) / Ex(\mathbf{p}_G) \end{aligned}$$

  $\mathit{Dep}(\mathbf{p}_G)$ are propositions that goals mentioned after $\mathbf{p}_G$ in Δ_G may be dependent on. We intersect with $\overline{M}(I, \mathbf{p}_G)$ to ensure the subproblem associated with a $\mathbf{p}_G$ in Δ_G is not trivially unsolvable given available mutex information. We remove all elements of $Ex(\mathbf{p}_G)$, as these can only change their polarity once, so we do not enforce that they are returned to their starting polarity.

- Each element $\mathbf{p}_G \in \Delta_G$ then gives a subproblem $\langle X_{\mathbf{p}_G}, A_{\mathbf{p}_G}, I_{\mathbf{p}_G}, G_{\mathbf{p}_G} \rangle$ where:

$$\begin{aligned} X_{\mathbf{p}_G} &= F(\mathbf{p}_G) \\ A_{\mathbf{p}_G} &= A{\downarrow}F(\mathbf{p}_G) \\ I_{\mathbf{p}_G} &= I{\downarrow}F(\mathbf{p}_G) \\ G_{\mathbf{p}_G} &= G{\downarrow}V(\mathbf{p}_G) \wedge I{\downarrow}Dep(\mathbf{p}_G) \end{aligned}$$

Each subproblem is passed to a PDR planning process, with such processes run in parallel to completion. If a plan is found for each subproblem, and the concatenation of those is validated as a concrete plan, then PD-PDR terminates having successfully found that concrete plan.

5.2.3 Merging on Plan Failure

The concatenation operation however may fail to produce a valid plan. This only happens due to exclusions or mutex – e.g., two subproblem plans seek to modify the truth value of a proposition that can only be changed once. If concatenation fails to yield a valid plan, then a proposition not mentioned in the subproblem goal due to an exclusion, or mutex, is identified as *problematic*. The problematic proposition is identified by first attempting to execute the concatenation of subproblem plans, which will fail. The problematic proposition is then the first unmet precondition of actions in the plan. PD-PDR then enters a new iteration, creating a corresponding new Γ by contracting vertices, and maybe adding propositions to the labelling of vertices, from the Γ used in this iteration. The details of the contractions are described below:

The LADG vertex the problematic proposition labels is contracted with all LADG vertices labelled with propositions in a mutex relationship with that problematic proposition. Furthermore, the vertex resulting from this contraction is contracted with all its descendants. This final contracted vertex is labelled with the union of the labels of all contracted vertices.

Should the above operation not result in two or more vertices being contracted—e.g., the vertex labelled with the problem proposition has no descendants or propositions in mutex relationships—then the vertex labelled with the problematic proposition and all its parents are contracted together. This ensures that when concatenation fails, the graph derived for the next iteration has fewer vertices than the Γ used in this iteration.

5.2.4 Unsolvable Subproblems

Before getting to the concatenation stage, it may be the case that one or more subproblems does not have a solution. If a subproblem is unsatisfiable, and its goal condition is a subformula of the concrete goal—as happens when $Dep(\mathbf{p}_G)$ is empty—then the concrete problem is unsatisfiable. Otherwise (if the concrete problem is not found to be unsatisfiable overall), then the LADG is contracted prior to a successive iteration being performed. The contractions are performed as follows: For each unsatisfiable subproblem, all vertices in the LADG whose labels mention any proposition in the subproblem are contracted to a single vertex. Additionally, for each unsatisfiable subproblem, should the above operation not result in two or more vertices being contracted, then the sole vertex labelled with propositions from the subproblem is contracted with all its parents. The above described contractions are carried out one unsatisfiable subproblem at a time, in the order that their corresponding $\mathbf{p}_G$ occurs in Δ_G.

Two example executions of PD-PDR are provided in Examples 7 and 8.

Example 7 (Performing PD-PDR on a Logistics Problem) *Consider the running Logistics example problem from Example 1/4.* PD-PDR *first takes this concrete planning problem and produces a PSDG, which shows how particular propositions depend on other propositions. This is used to produce a LADG* Γ *for the initial iteration, which groups propositions into components, and shows how the components depend on each other. The PSDG and corresponding LADG is shown in Figure 5.2.*

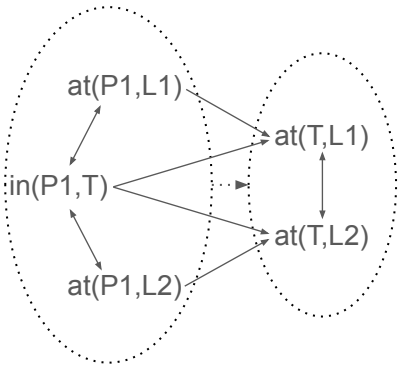


Figure 5.1: Logistics problem specific dependency graph, depicted with a LADG whose vertices, indicated by dashed ovals, correspond to SCCs of the PSDG.

$\Gamma = (\{\mathbf{p}_P, \mathbf{p}_T\}, \{(\mathbf{p}_P, \mathbf{p}_T)\})$ *where:*

$$\begin{aligned} V(\mathbf{p}_P) &= \{at(P1, L1), at(P1, L2), in(P1, T)\} \\ V(\mathbf{p}_T) &= \{at(T, L1), at(T, L2)\} \end{aligned}$$

No propositions which mention P2 are included, as while they are in the dependency

graph, they are not in the PSDG, as $P2$ is not mentioned in the goal.

$\Delta_G = [\mathbf{p}_P, \mathbf{p}_T]$ is then constructed, which represents a list of subgoals, each of which indicate a corresponding subproblem. This contains all the vertices in the current LADG which contain propositions mentioned in the concrete goal. The order is one that is consistent with the partial order imposed by the edges of the LADG. All SCCs are mentioned as they all include propositions mentioned in G. $\mathbf{p}_P$ occurs before $\mathbf{p}_T$ as there is a path $\mathbf{p}_P \overset{\Gamma}{\rightsquigarrow} \mathbf{p}_T$.

PD-PDR *then constructs a series of subproblems, each of which describes the problem of computing part of a concrete plan. To construct these subproblems, for each $\mathbf{p}_G \in \Delta_G$, $F(\mathbf{p}_G)$ is computed. This is the set of most (but possibly not all) of the propositions needed to plan for the $\mathbf{p}_G$ portion of the goal. $F(\mathbf{p}_G)$ is the union of all propositions mentioned in $\mathbf{p}_G$, with all propositions mentioned in descendants of $\mathbf{p}_G$ in Γ. By taking the descendants, this ensures all propositions that are depended upon are included in the subproblem propositions, and only the propositions which are not depended upon are excluded.*

$$\begin{aligned} F(\mathbf{p}_P) &= \{at(P1, L1), at(P1, L2), in(P1, T), \\ &\quad at(T, L2), at(T, L2)\} \\ F(\mathbf{p}_T) &= \{at(T, L1), at(T, L2)\} \end{aligned}$$

Next, the exclusions, $\mathrm{Ex}(\mathbf{p}_G)$ *are considered. These are propositions which can only change polarity once, often representing a finite resource. In this example there are no such propositions.*

$$\begin{aligned} \mathrm{Ex}(\mathbf{p}_P) &= \{\} \\ \mathrm{Ex}(\mathbf{p}_T) &= \{\} \end{aligned}$$

Each subproblem goal is the conjunction of a section of the concrete goal being achieved in that subproblem, with a portion of the initial state. As each subproblem initial state is a subformula of the concrete problem initial state, each subproblem goal condition contains the conjunction of the initial states of all subsequent subproblems. To do this, PD-PDR *calculates dependants* $\mathrm{Dep}(\mathbf{p}_G)$*. $\mathrm{Dep}(\mathbf{p}_G)$ is somewhat similar to $F(\mathbf{p}_G)$, but contains all the propositions relevant to later subproblems indicated by Δ_G. For the last subproblem, $\mathrm{Dep}(\mathbf{p}_G)$ will always be $\emptyset$.*

$$\begin{aligned} \mathrm{Dep}(\mathbf{p}_P) &= \{at(T, L1), at(T, L2)\} \\ \mathrm{Dep}(\mathbf{p}_T) &= \{\} \end{aligned}$$

The two subproblems are then:

- $\langle X_{\mathbf{p}_P}, A_{\mathbf{p}_P}, I_{\mathbf{p}_P}, G_{\mathbf{p}_P} \rangle$ *where:*

$$\begin{array}{lcl} X_{\mathbf{p}_P} & = & \{at(P1, L1), at(P1, L2), in(P1, T), \\ & & at(T, L1), at(T, L2)\} \\ A_{\mathbf{p}_P} & : & \textit{Can be described by the actions:} \\ & & \textit{Load } P1 \textit{ into } T \textit{ from } L1 \textit{ or } L2, \\ & & \textit{Unload } P1 \textit{ from } T \textit{ to } L1 \textit{ or } L2, \\ & & \textit{Drive } T \textit{ between } L1 \textit{ and } L2 \\ I_{\mathbf{p}_P} & = & at(P1, L1) \wedge \neg at(P1, L2) \wedge \neg in(P1, T) \wedge \\ & & at(T, L1) \wedge \neg at(T, L2) \\ G_{\mathbf{p}_P} & = & at(P1, L2) \wedge at(T, L1) \end{array}$$

- $\langle X_{\mathbf{p}_T}, A_{\mathbf{p}_T}, I_{\mathbf{p}_T}, G_{\mathbf{p}_T} \rangle$ *where:*

$$\begin{array}{lcl} X_{\mathbf{p}_T} & = & \{at(T, L1), at(T, L2)\} \\ A_{\mathbf{p}_T} & : & \textit{Can be described by the actions:} \\ & & \textit{Drive } T \textit{ between } L1 \textit{ and } L2 \\ I_{\mathbf{p}_T} & = & at(T, L1) \wedge \neg at(T, L2) \\ G_{\mathbf{p}_T} & = & at(T, L2) \end{array}$$

Each of which is solved independently in parallel. Two corresponding example subproblem plans are then:

- *Load* $P1$ *into* T *at* $L1$*;*
  Drive T *to* $L2$*;*
  Unload $P1$ *from* T *at* $L2$*;*
  Drive T *to* $L1$
- *Drive* T *to* $L2$

When concatenated, these two plans for the subproblems make a valid plan for the concrete problem. In this example, there is no need to start another iteration of the algorithm.

Example 8 (Performing PD-PDR **on a Logistics Problem with Limited Fuel)** *This example is designed to show how subproblems are restructured when the concatenation of subproblem plans fails to produce a valid plan because of exclusions. Consider a modification to the running Logistics problem presented in Example 1/4, where the truck can only move once. In this new example, there are two locations* $L1$ *and* $L2$*, a single package* P*, a single truck* T*, and two units of fuel for the truck,* $fuel1(T)$ *and* $fuel2(T)$*. The package can be (un)loaded into the truck at either*

location. The truck can only drive if it has fuel, and driving it removes one of its units of fuel.[1] *The initial state has the truck and the package at* $L1$*, and both units of fuel able to be used. The goal condition has the truck and package at* $L2$*. The dependency graph and corresponding LADG of this problem is shown in Figure 5.2.*

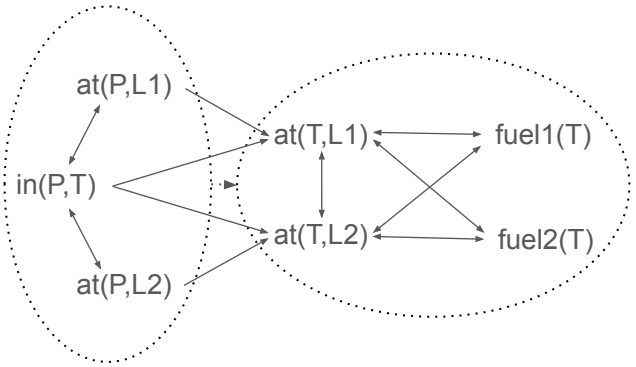


Figure 5.2: Fuel limited Logistics problem dependency graph, depicted with a LADG whose vertices, indicated by dashed ovals, correspond to SCCs of that dependency graph.

We perform PD-PDR *on this problem. The first iteration proceeds similarly to Example 7. We first compute* $\Gamma = (\{\mathbf{p}_P, \mathbf{p}_T\}, \{(\mathbf{p}_P, \mathbf{p}_T)\})$ *where:*

$$\begin{aligned} V(\mathbf{p}_P) &= \{at(P, L1), at(P, L2), in(P, T)\} \\ V(\mathbf{p}_T) &= \{at(T, L1), at(T, L2), fuel1(T), fuel2(T)\} \end{aligned}$$

We then compute: $\Delta_G = [\mathbf{p}_P, \mathbf{p}_T]$*. Unlike in Example 7 there are now exclusions for the fuel.*

$$\begin{aligned} F(\mathbf{p}_P) &= \{at(P, L1), at(P, L2), in(P, T), \\ &\quad at(T, L2), at(T, L2), fuel1(T), fuel2(T)\} \\ F(\mathbf{p}_T) &= \{at(T, L1), at(T, L2), fuel1(T), fuel2(T)\} \\ \mathrm{Ex}(\mathbf{p}_P) &= \{fuel1(T), fuel2(T)\} \\ \mathrm{Ex}(\mathbf{p}_T) &= \{fuel1(T), fuel2(T)\} \\ \mathrm{Dep}(\mathbf{p}_P) &= \{at(T, L1), at(T, L2)\} \\ \mathrm{Dep}(\mathbf{p}_T) &= \{\} \end{aligned}$$

Similarly to Example 7, but with the addition of the fuel, the two subproblems are then:

[1] In the encoding, there are separate actions corresponding to the use of each unit of fuel.

- $\langle X_{\mathbf{p}_P}, A_{\mathbf{p}_P}, I_{\mathbf{p}_P}, G_{\mathbf{p}_P} \rangle$ *where:*

$$\begin{aligned} X_{\mathbf{p}_P} &= \{at(P, L1), at(P, L2), in(P, T), \\ &\quad at(T, L1), at(T, L2) \\ &\quad fuel1(T), fuel2(T)\} \\ A_{\mathbf{p}_P} &: \textit{Can be described by the actions:} \\ &\quad \textit{Load } P \textit{ into } T \textit{ from } L1 \textit{ or } L2, \\ &\quad \textit{Unload } P \textit{ from } T \textit{ to } L1 \textit{ or } L2, \\ &\quad \textit{Drive } T \textit{ between } L1 \textit{ and } L2, \textit{ using a unit of fuel} \\ I_{\mathbf{p}_P} &= at(P, L1) \wedge \neg at(P, L2) \wedge \neg in(P, T) \wedge \\ &\quad at(T, L1) \wedge \neg at(T, L2) \\ &\quad fuel1(T) \wedge fuel2(T) \\ G_{\mathbf{p}_P} &= at(P, L2) \wedge at(T, L1) \end{aligned}$$

- $\langle X_{\mathbf{p}_T}, A_{\mathbf{p}_T}, I_{\mathbf{p}_T}, G_{\mathbf{p}_T} \rangle$ *where:*

$$\begin{aligned} X_{\mathbf{p}_T} &= \{at(T, L1), at(T, L2) \\ &\quad fuel1(T), fuel2(T)\} \\ A_{\mathbf{p}_T} &: \textit{Can be described by the actions:} \\ &\quad \textit{Drive } T \textit{ between } L1 \textit{ and } L2, \textit{ using a unit of fuel} \\ I_{\mathbf{p}_T} &= at(T, L1) \wedge \neg at(T, L2) \\ &\quad fuel1(T) \wedge fuel2(T) \\ G_{\mathbf{p}_T} &= at(T, L2) \end{aligned}$$

As in example 7, two corresponding example subproblem plans are then:

- *Load* $P1$ *into* T *at* $L1$*;*
  Drive T *to* $L2$*;*
  Unload $P1$ *from* T *at* $L2$*;*
  Drive T *to* $L1$
- *Drive* T *to* $L2$

When concatenated, these two plans do not make a valid plan for the concrete problem, as the truck is driven three times, and there are only two units of fuel.[2] *As a result, one of the units of fuel is identified as the problematic proposition. This results in a new iteration being started. The LADG of this new iteration only has a single vertex, the result of the combination of the two vertices of the LADG from the*

[2]Note that in a different instance where two units of fuel are available and each subproblem plan only consumes one, the concatenation could still fail if they use the same one.

first iteration. This sole vertex results in a sole "subproblem", which corresponds to the concrete problem.

5.3 PD-PDR is Sound and Complete

PD-PDR will always have a finite number of iterations, as each iteration decreases the number of subproblems, to a minimum of one. Each iteration consists of a set number of PDR processes, and so takes a finite amount of time, and as such PD-PDR terminates.

When there is only a single subproblem, the answer to that problem is the same as that of the concrete problem. These problems will be identical, except that propositions that the goal does not depend on are trimmed, so will not affect the existence of a plan.

When there are multiple subproblems, PD-PDR only returns a plan when a concatenation of subproblem plans yields a valid plan, as per the plan verifier. When there are multiple subproblems, PD-PDR only declares a plan does not exist when a subproblem is unsolvable and the subproblem goal is a subformula of the concrete problem's conjunctive goal. The subproblem goal represents a portion of the goal, and the rest of the subproblem represents everything that could be needed to achieve that goal segment, so if this subproblem returns *False*, then the concrete problem has no solution.

PD-PDR is sound and complete as it terminates, and any returned answer is correct. Note that even if obligation rescheduling is disabled, PD-PDR may not return plans of minimal length.

5.4 Experimental Evaluation

We here implement and evaluate the performance of PS-PDR and PD-PDR on a range of benchmarks, against a range of comparative solvers.

5.4.1 Setup

We implemented the PS-PDR and PD-PDR planners in `C++`, using the Madagascar codebase for parsing and preprocessing (Rintanen 2012). We also implemented a serial PDR planner, called *PDR-Serial* (PDR-S), and the distributed PDR portfolio PDR-P, introduced in Section 4.3.Lingeling is used as the incremental SAT solver in all our systems.

Our motivations for implementing the comparison algorithms, PDR-S and PDR-P planners, are threefold:

- We isolate the reasons for performance gaps, employing the same data structures and SAT decision procedure across all implementations.
- All algorithms we implemented feature obligation rescheduling, which is an algorithmic choice in PDR that is generally motivated in planning benchmarks, and not generally supported by existing PDR portfolios.
- We are able to natively parse and process planning benchmarks in PDDL, which is not possible using existing PDR portfolios that are implemented for hardware and software model checking.

Finally, our experimentation also evaluates PDRplan (Suda 2014), an implementation of serial PDR for planning that uses bespoke decision procedures for evaluating obligations. We include PDRplan as it is an important historical precedent and relevant contribution, however comparing it with our solvers is not a like for like comparison. Its bespoke decision procedure, differing parsing abilities and internal representations, makes it a significantly different procedure, while still being a PDR solver. We use PDR-S as our serial PDR baseline to showcase differences in algorithms, instead of implementation details.

We evaluate planners over the comprehensive benchmark set from (Rintanen 2012), which is compatible with Madagascar. These are satisfiable benchmarks from International Planning Competitions (IPCs). We additionally use the unsatisfiable benchmarks from the 2016 IPC Unplannability track.[3] The original competition had satisfiable and unsatisfiable instances, here we just use the unsatisfiable instances. We exclude reporting on some unsatisfiable domains, namely: Bag-Barman, Bag-Gripper, Over-TPP and Sliding-Tiles, because no systems evaluated are able to produce proofs of unsatisfiability in those domains under our timeout and memory limits.

We use two systems for evaluation. An Intel(R) Xeon(R) Platinum 8274 CPU host with 198GBs memory was used for the satisfiable benchmarks. An Intel(R) Xeon(R) Gold 6252 CPU host with 187GBs memory was used for the unsatisfiable benchmarks. The use of two separate hosts was based solely on the availability of computing infrastructure.

All problem instances were run with a 30-minute (1800s) timeout. Each PS-PDR/PDR-P run used 48 cores (one thread per core), with 47 workers and one orchestrator. Our choice of 48 core configurations is based on available computing

[3]Domains available at github.com/AI-Planning/unsolve-ipc-2016

infrastructure during our experimentation, and is not indicative of where to expect optimal performance. In the case of PD-PDR, our implementation first decomposes the problem using a single core process to create a list of subproblems. Subproblems are solved independently in parallel, using our PDR-S implementation. As PD-PDR can produce many subproblems, the runtimes reported for PD-PDR are a simulated runtime where the number of simulated cores is the number of subproblems, all with sufficient memory. In other words, we simulate having enough cores for each subproblem, where each core has the memory of the entire computer. However in practice, these additional simulated resources rarely impact the performance. PD-PDR candidate concrete plan validity is determined using VAL (Howey et al. 2004). We only evaluate PD-PDR on satisfiable planning benchmarks that we know are susceptible to compositional analysis. We only evaluate PD-PDR under these circumstances, as otherwise PD-PDR will function as PDR-S, with some additional overheads

5.4.2 Results

In total, 58 domains are reported on, with PS-PDR, PDR-P and PDR-S able to parse all of them. There are 16 domains that PDRPlan is not able to parse. PS-PDR is able to solve 1481 problem instances, PDR-P can solve 1455, PDR-S can solve 1421 and PDRplan can solve 990.[4] There are 12 satisfiable domains susceptible to the decomposition used by PD-PDR where PD-PDR produces a plan during an iteration which has more than one subproblem. In our reporting, when considering PD-PDR, only these 12 domains are reported on. Of the 12 domains reported on here, PD-PDR has the best or equal best coverage in 10 domains. PD-PDR can solve 331 of the 346 instances that are amenable to decomposition.

We note that when comparing all systems, and only considering the 42/58 domains that PDRPlan can parse, PDRplan solves the largest number of problems. However PDRplan has the sole best coverage in only 12/42 domains, and the best average time in only 19/42 domains. Because of this, while having the best coverage, we do not consider PDRplan to be the clearly superior solver.

Scatterplots of the time taken by each solver compared to the baseline PDR-S, to solve each problem is provided in Figures 5.5-5.7. Note the number of instances solved by PD-PDR in Figure 5.7 is relatively low, because we only report instances amenable to decomposition. Figures 5.3 and 5.4 feature the cumulative number of problems solved, as a function of time, for each solver. Figure 5.3 reports on all instances, but does not include PD-PDR, as PD-PDR is not suitable to all

[4]Low coverage in the case of PDRplan is exacerbated by parser issues.

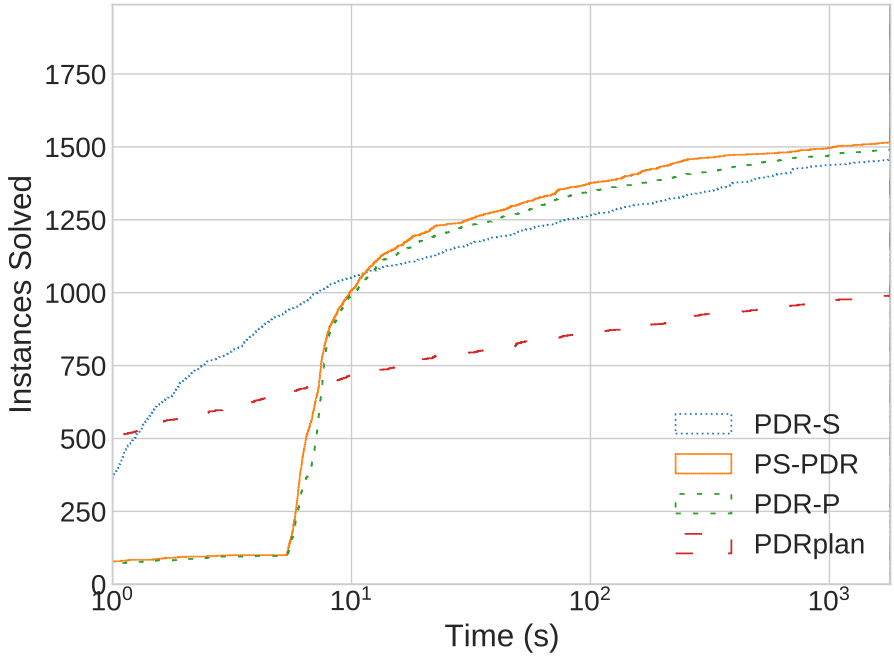


Figure 5.3: Cumulative number of instances solved. Reporting on all instances. Does not report on PD-PDR.

instances. Conversely figure 5.4 reports on PD-PDR, but only reports on satisfiable instances amenable to decomposition.

We also supply full coverage data in Tables 5.1-5.3. These tables break the benchmarks up into decomposable satisfiable problems (Table 5.1), non-decomposable satisfiable problems (Table 5.2) and unsatisfiable problems (Table 5.3). The first column reports the domain name, with the total number of problems. In Table 5.1, column $\overline{\#CPUs}$ reports the maximum number of CPUs allocated to a PD-PDR problem in the domain. Domains that do not include a result for PD-PDR and $\overline{\#CPUs}$ are not decomposable. System-headed columns, e.g., the column headed "PS-PDR", report the number of instances solved for each domain (coverage) along with the average time taken to solve a problem in the domain. When the solver times out or runs out of memory on a problem, the timeout of $1800s$ is noted instead as the time taken on that problem. The best coverage and average runtime for each domain is in bold. When PDRplan is unable to process problems from a domain, "N/A" is instead reported for that domain.

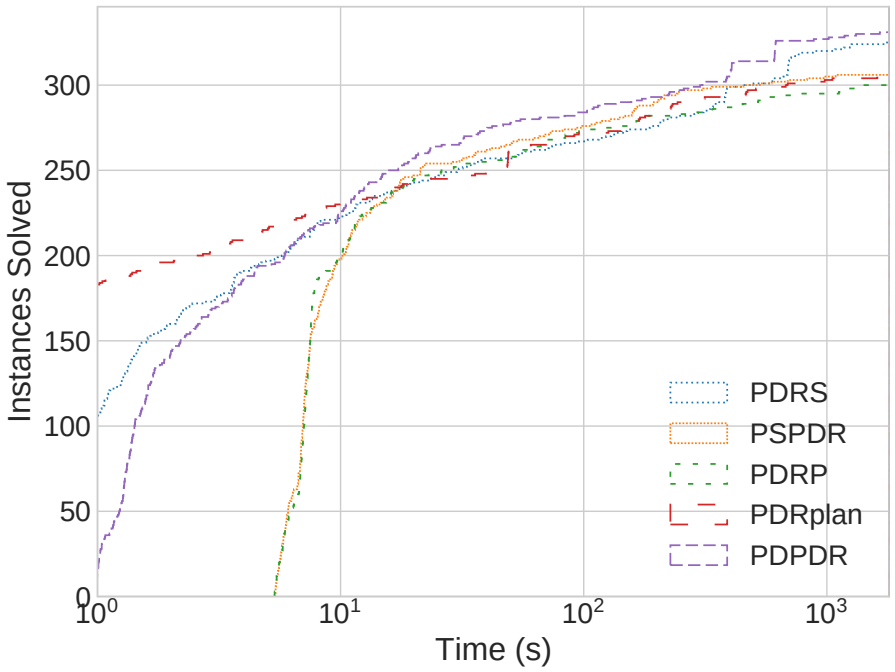


Figure 5.4: Cumulative number of instances solved. Reporting only on satisfiable instances amenable to decomposition. Includes reporting on PD-PDR.

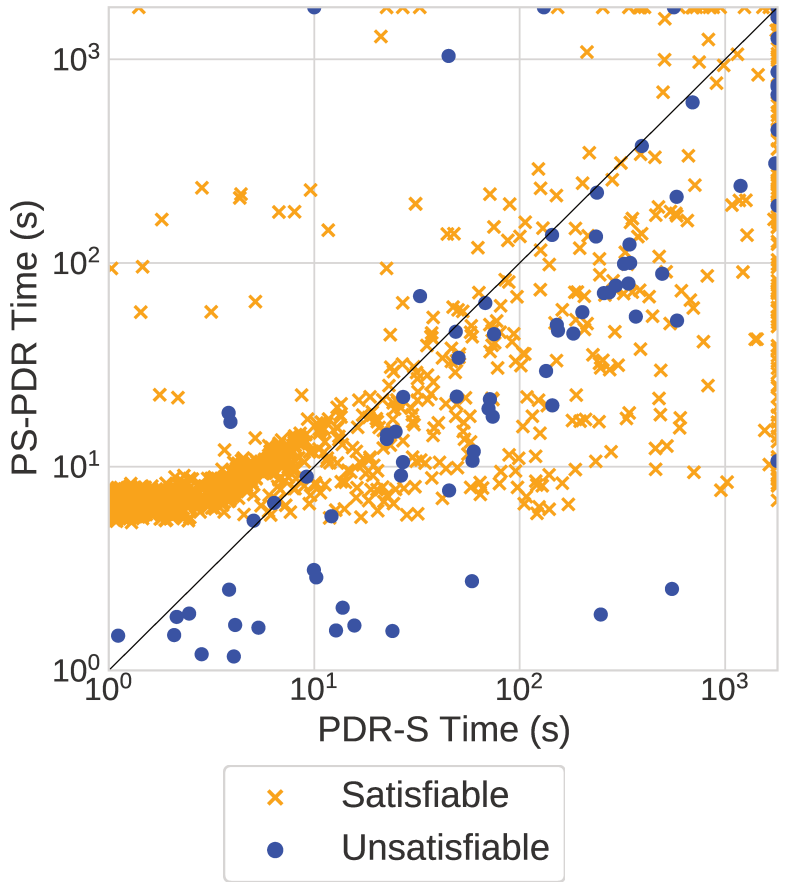


Figure 5.5: Scatterplot Comparing Problem Runtimes Between PS-PDR and the Baseline PDR-S. Reports on all instances.

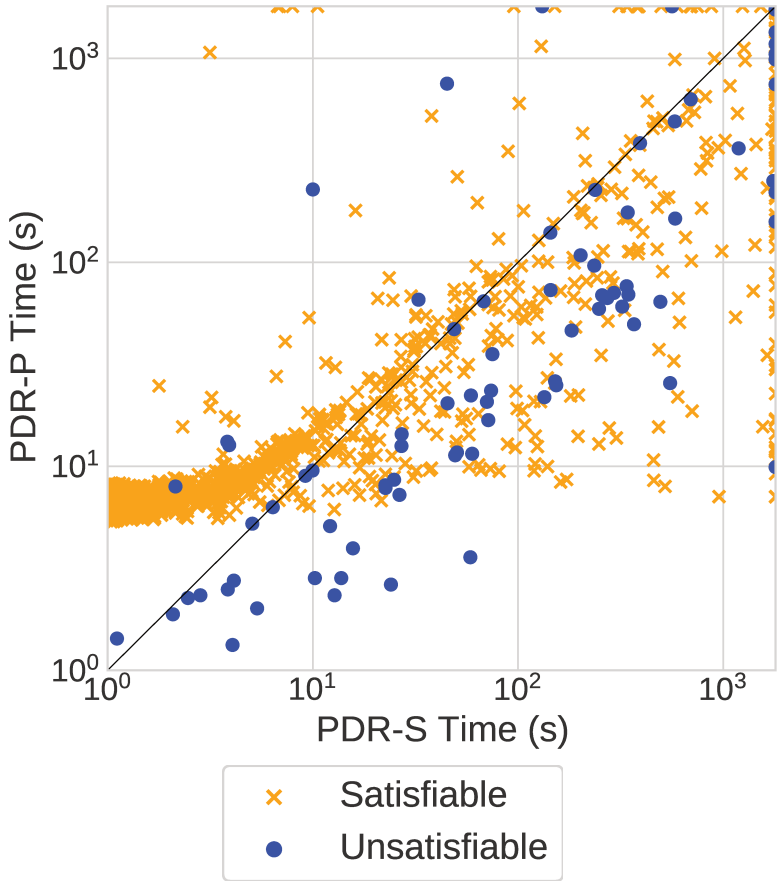


Figure 5.6: Scatterplot Comparing Problem Runtimes Between PDR-P and the Baseline PDR-S. Reports on all instances.

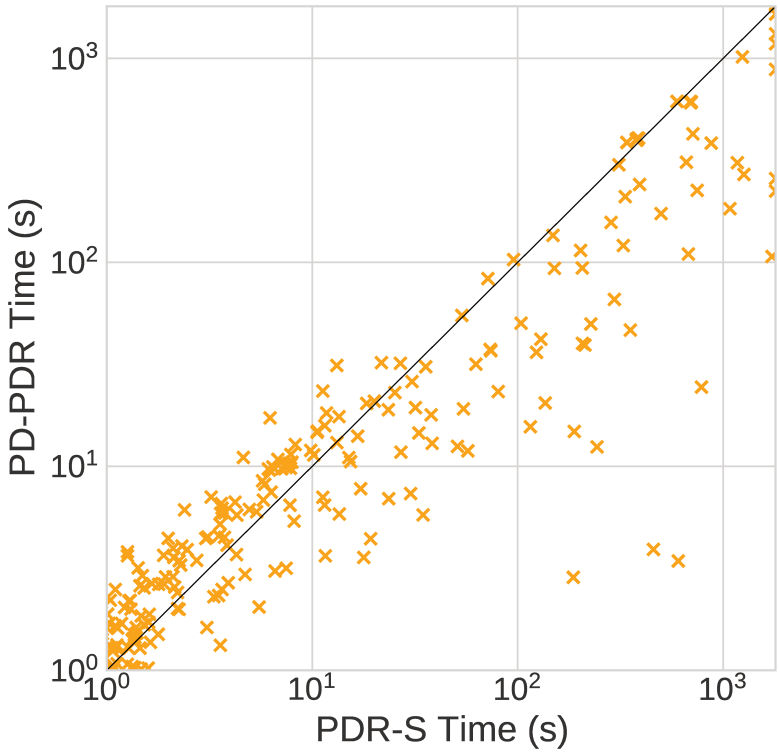


Figure 5.7: Scatterplot Comparing Problem Runtimes Between PD-PDR and the Baseline PDR-S. Reports only on satisfiable instances amenable to decomposition.

Domain (#Instances)	PS-PDR	PDR-P	PDR-S	PDRplan	PD-PDR	$\overline{\#CPUs}$
98-LOGISTICS (30)	**30 (50.5)**	29 (119.7)	29 (166.1)	14 (1001.5)	**30** (74.0)	45
98-MOVIE (30)	**30** (61.6)	**30** (6.8)	**30** (0.8)	**30 (0.0)**	**30** (1.4)	7
02-DRIVERLOG (20)	**20** (8.1)	**20** (14.6)	**20** (17.5)	17 (286.4)	**20 (2.7)**	26
02-ZENO (20)	**20** (10.4)	**20** (19.1)	**20** (11.2)	**20** (28.0)	**20 (5.1)**	27
04-PSR-SMALL (50)	**50** (6.7)	**50** (6.8)	**50** (0.9)	**50 (0.0)**	**50** (1.4)	3
04-SATELLITE (36)	28 (449.3)	28 (451.6)	28 (500.2)	28 (461.4)	**30 (346.3)**	187
06-ROVERS (40)	**40 (40.4)**	39 (154.3)	38 (174.6)	**40** (65.6)	39 (84.0)	69
06-TPP (30)	**30 (30.9)**	**30** (90.4)	**30** (132.1)	**30** (119.0)	**30** (37.4)	20
08-CYBER-SECURITY (30)	8 (1321.9)	8 (1321.8)	**30** (396.3)	**30 (30.5)**	**30** (378.5)	2
11-SCANALYZER (20)	**19 (168.4)**	**19** (229.4)	17 (277.2)	18 (239.7)	18 (271.7)	19
11-VISITALL (20)	11 (939.5)	8 (1267.1)	13 (837.3)	9 (1094.4)	**14 (721.4)**	2499
11-WOODWORKING (20)	**20** (11.5)	**20** (11.9)	**20 (6.8)**	N/A	**20** (10.0)	32
Total (346)	306	301	325	286	**331**	

Table 5.1: Coverage and average runtimes for satisfiable decompositional benchmarks.

Domain (#Instances)	PS-PDR	PDR-P	PDR-S	PDRplan
98-ASSEMBLY-ADL (24)	**24** (5.9)	**24** (6.7)	**24 (1.0)**	N/A
98-GRID (5)	**5 (21.1)**	**5** (76.1)	**5** (119.7)	**5** (36.8)
98-GRIPPER (21)	**21** (7.4)	**21** (7.7)	**21** (1.1)	**21 (0.0)**
98-MPRIME (20)	**20** (34.7)	**20 (27.3)**	19 (117.2)	N/A
98-MYSTERY (19)	18 (105.9)	**19 (11.3)**	18 (100.3)	15 (394.6)
00-BLOCKS (102)	96 (132.2)	**102 (18.3)**	96 (170.2)	101 (46.2)
00-ELEV-SIMPLE (150)	**150** (7.1)	**150** (7.1)	**150 (1.2)**	N/A
00-FREECELL (60)	**59** (123.0)	**59** (152.5)	56 (242.4)	**59 (62.7)**
00-SCHED-ADL (150)	**150** (10.3)	**150** (8.5)	**150 (4.1)**	N/A
02-DEPOTS (22)	**22 (8.0)**	**22** (8.7)	21 (83.9)	19 (326.5)
02-SATEL-ADL (20)	**20 (8.8)**	**20** (12.4)	**20** (17.0)	N/A
04-AIRPORT (50)	25 (932.6)	22 (1017.3)	20 (1087.0)	**41 (372.3)**
04-AIRPORT-ADL (50)	29 (795.4)	**35 (654.4)**	28 (852.6)	N/A
04-OPT-TELE (14)	5 (1177.7)	5 (1189.0)	4 (1386.6)	**14 (18.2)**
04-OPT-TELE-ADL (48)	6 (1578.7)	**8 (1541.9)**	6 (1627.3)	N/A
04-PHILOS (29)	**29** (7.5)	**29** (18.3)	**29** (36.0)	**29 (1.6)**
04-PHILOS-ADL (48)	**45 (156.9)**	36 (487.6)	33 (584.7)	N/A
04-PIPE-NOTANK (50)	45 (201.1)	45 (223.6)	43 (354.5)	**47 (187.2)**
04-PIPE-TANK (50)	39 (506.2)	38 (552.4)	28 (947.2)	**40 (485.1)**
06-PATHWAYS (30)	**30** (6.9)	**30** (7.9)	**30** (2.3)	**30 (1.1)**
06-STORAGE (30)	29 (140.9)	20 (621.9)	21 (600.0)	**30 (19.9)**
06-TRUCKS (30)	10 (1265.2)	9 (1286.1)	8 (1322.7)	**27 (305.8)**
06-TRUCKS-ADL (30)	**10** (1240.0)	**10 (1214.7)**	**10** (1265.8)	N/A
08-OPENSTA-ADL (30)	**30 (29.5)**	**30** (61.1)	**30** (44.6)	N/A
11-BARMAN (20)	3 (1659.0)	1 (1726.3)	0 (1800.0)	**7 (1346.8)**
11-ELEVATORS (20)	12 **(742.2)**	7 (1242.9)	5 (1385.8)	**13** (773.7)
11-FLOORTILE (20)	**20 (7.9)**	**20** (27.8)	19 (116.1)	15 (574.3)
11-OPENSTACKS (20)	16 (631.6)	15 (591.2)	16 (580.0)	**20 (216.7)**
11-PARCPRINTER (20)	**20** (6.3)	**20** (7.4)	**20 (1.1)**	**20** (3.1)
11-PARKING (20)	**3 (1698.1)**	0 (1800.0)	0 (1800.0)	N/A
11-SAT-PEGSOL (20)	**20** (6.4)	**20** (6.6)	**20** (2.4)	**20 (1.0)**
11-SOKOBAN (20)	**9 (1200.2)**	8 (1286.0)	7 (1412.9)	5 (1443.9)
11-TIDYBOT (20)	**16 (90.0)**	**16** (101.9)	15 (227.5)	N/A
11-TRANSPORT (20)	5 **(1369.6)**	3 (1639.6)	1 (1720.5)	**8** (1475.8)
Total (1302)	**1051**	1029	979	620

Table 5.2: Coverage and average runtimes for satisfiable non-decompositional benchmarks.

Domain (#Instances)	PS-PDR	PDR-P	PDR-S	PDRplan
16-UNSAT-BAG-TRANSPORT (29)	**12** (1092.6)	**12 (1080.9)**	9 (1297.6)	N/A
16-UNSAT-BOTTLENECK (25)	**25 (1.0)**	**25** (3.5)	**25** (12.7)	20 (385.3)
16-UNSAT-CAVE-DIVING (25)	5 (1444.7)	**6 (1441.9)**	5 (1444.7)	5 (1460.0)
16-UNSAT-CHESSBOARD-PEBBLING (23)	**4 (1516.9)**	**4** (1520.6)	3 (1568.6)	3 (1576.1)
16-UNSAT-DIAGNOSIS (20)	18 (247.7)	19 (171.3)	**20 (49.2)**	15 (451.8)
16-UNSAT-DOCUMENT-TRANSFER (20)	**11 (884.6)**	**11** (886.2)	**11** (893.3)	4 (1467.9)
16-UNSAT-NOMYSTERY (20)	10 (1073.5)	10 (1069.6)	6 (1272.7)	**15 (665.8)**
16-UNSAT-OVER-NOMYSTERY (24)	**6 (1408.1)**	5 (1442.4)	5 (1470.0)	N/A
16-UNSAT-OVER-ROVERS (20)	**19 (215.2)**	**19** (258.9)	17 (482.4)	17 (378.6)
16-UNSAT-PEGSOL-ROW5 (15)	3 (1321.1)	3 (1320.6)	3 (1321.7)	**4 (1320.1)**
16-UNSAT-TETRIS (20)	**5** (1372.2)	**5 (1367.1)**	**5** (1439.6)	N/A
16-UNSAT-PEGSOL (24)	**16** (728.8)	**16** (695.2)	14 (771.1)	**16 (690.5)**
Total (245)	124	**125**	117	84

Table 5.3: Coverage and average runtimes for unsatisfiable benchmarks.

5.5 Conclusion

We here present a parallel PDR-based solver which decomposes a problem based on its compositional structure. This produces a collection of subproblems, which are solved independently in parallel, and a plan for the concrete problem is created as a concatenation of subproblem plans.

We find PD-PDR (when it is applicable) and PS-PDR to be the fastest solvers. Due to the startup cost of MPI, PS-PDR/PDR-P solved fewer instances in the first $\sim$10 seconds than the other solvers.

In 39 cases, PS-PDR exhausted the memory of the host, and PDR-P did in 40 cases, no other solver did this. However we note that PD-PDR has the unfair advantage that each subproblem is given the entire memory of the system, and the results are reported as if all subproblems were able to be run concurrently on the same machine. In spite of this in only two domains where there any instances which had more subproblems than there were available cores for our experiment.

Chapter 6

PDR for FOND

We develop, implement and evaluate a new FOND algorithm based on PDR which we call FOND-PDR. Although the atomic SAT operations in classical PDR are about whether a state can, or cannot be advanced to the goal, at a macro level that PDR algorithm proceeds by excluding the possibility of a plan of length k, for increasing values of k. With an analogous condition in mind, we adapt PDR to have a new invariant condition suitable for non-deterministic execution. Here, a SAT query is successful iff one outcome advances towards the goal, and all other outcomes keep the agent within a k-horizon from the goal. Thus, iteratively for increasing values of k, and representing the policy as a cyclic AND/OR graph, our approach searches in the space of graphs where executions ending in a goal state can traverse at most k or fewer distinct states.

An appealing property of classical PDR is the ability to detect unreachability, and our adaptation of PDR to FOND planning exhibits this. As with classical PDR, our approach is in theory sound and complete, specifically because in the worst case we can exclude the possibility of a policy when k is a completeness threshold (CT). Moreover, the reachability information collected by the algorithm allows easily discovering that the goal cannot be reached, and can thereby determine that a FOND problem has no solution. Finally, our algorithm features a FOND policy computation that operates on the AND/OR graph of states discovered by PDR evaluations. It is thereby able to produce and report policies as they are found. We evaluated our approach on both solvable and unsolvable FOND planning instances.

Adapting PDR to work in a FOND setting includes modifying many of its internal workings, most notably:

- The SAT call.
- The criterion for stopping, both when a plan does and doesn't exist.
- The queue.

Otherwise we carry over many of the concepts from classical PDR.

We remind the reader of a definition first given in Section 2.3. For a policy π, we write $\lceil \pi \rceil_s$ to denote the maximum possible number of distinct states encountered in an execution from s to the goal, not including s. It is important to note that if a state is encountered multiple times in an execution, it is only counted once.

6.1 FOND-PDR Structure

A main difference between classical PDR and FOND-PDR is that the former is looking for paths from states (including the initial state) to a goal state, and each state and action determine a unique successor state. FOND-PDR however creates a cyclic and branching plan, in which for a single action a state may have more than one successor state when the action is non-deterministic with multiple outcomes. The overall goal is to identify a set S of states and a policy π so that, first, there is a path from each $s \in S$ to a goal state with the actions given by π, and, second, for every $s \in S$, all outcomes of the action $\pi(s)$ will generate another state in S. This is exactly what is required by *strong cyclic plans* (Cimatti et al. 2003).

Similarly to PDR, FOND-PDR maintains and iteratively refines layer CNF formula which represent an over-approximations of reachability information. Each formula $\mathcal{L}_i$ satisfies the invariant condition: If $s \not\models \mathcal{L}_i$, there is no policy such that $\lceil \pi \rceil_s \leq i$ (Theorem 2). Like in classical PDR, initially, $\mathcal{L}_0 \equiv G$ and all other layers are the vacuously satisfied empty CNF formula, and if a clause is added to a layer at index i, then it is always included at the layers of all lower indices as well.

We also borrow the notion of an obligation from PDR as a pair $\langle s, i \rangle$ where s is a state and i is a layer index. We change how obligations are processed however. Specifically, we say an obligation $\langle s, i \rangle$ is *progressable* if there exists an action a, applicable to s, such that $\exists s' \in succ(s, a)$ $s.t.$ $s' \models \mathcal{L}_{i-1}$ and $\forall s' \in succ(s, a), s' \models \mathcal{L}_k$. The index k is in reference to the core FOND-PDR algorithm (Algorithm 4) and like in classical PDR changes over the course of the algorithm's execution, whenever a progressability test is made, it is always in reference to the current value of k. Like with classical PDR, this is done with a general purpose incremental SAT solver, but with a different encoding, the details of which are described in Section 6.4.

FOND-PDR maintains a queue as classical PDR does, but the queue's function is more complicated than that of classical PDR. We say a state s is *at* layer i in the queue if the obligation $\langle s, i \rangle$ is in the queue. When an obligation is taken from the queue to be processed, the queue also provides a set of *unauthorized actions* (UA). The specifics of why some actions are unauthorized is explained later. For now it suffices to say they are unauthorized so we do not repeat ourselves, exploring the

same state-action more than once. The *policy generator* is a component responsible for keeping track of which states/actions have been explored, and which states are known to have a policy and are therefore solved. The full working of this and the new queue, along with proofs they function as intended are provided later.

FOND-PDR iteratively proceeds by taking an obligation $\langle s, i \rangle$ along with a set of unauthorized actions from the queue (Line 9), and then tries to find an authorized action to progress this obligation. If such an action exists, then the outcomes are used to form obligations which are then added to the queue. Each outcome is added to the queue with the index of the lowest layer formula it satisfies – i.e – for each outcome o, the obligation $\langle succ(s, o), j \rangle$ is added to the queue, where j is the lowest j such that $succ(s, o) \vDash \mathcal{L}_j$. Additionally the originating obligation $\langle s, i \rangle$ is added back to the queue. If however no such action exists and the state cannot be progressed, what happens next depends on whether there are any unauthorized actions. If there are unauthorized actions, then the non-progressability of s may be due to these unauthorized actions, so the obligation is given back to the queue to be processed later when the actions are authorized.

If there are no unauthorized actions, then a reason is derived and used to constrain the layer information, identically to how this is done in classical PDR. We say a reason is *valid* as part of a layer formula if all states consistent with the reason are not progressable. Similarly we say a reason is *forward-valid* as part of a layer formula $\mathcal{L}_i$, if all states s consistent with the reason do not have a corresponding executable action with an outcome o such that $succ(s, o) \vDash \mathcal{L}_{i-1}$ – i.e are not progressable if $\mathcal{L}_k \equiv True$. For a set of reasons at a layer—possibly all reasons at the layer—we say they can be *pushed* to a layer j if they would be valid at layer j if they were used to constrain layer $j - 1$. To push a set of reasons to layer j is then to add them to all layers up to and including j. We define *forward-pushing* in an identical manner, replacing *valid* with *forward-valid*. We refer to forward-pushing and pushing Lines 4 and 6 respectively.

6.2 The Core Algorithm

We can now tie these ideas to the pseudo-code of Algorithm 4. First on Line 1, check is made to see if the initial state satisfies the goal condition, and if so then the problem is trivially satisfiable. The layer formulae are initialized (Line 2), setting $\mathcal{L}_0$ to be the goal, and all other layers to be the vacuously satisfied empty formula. After that, a loop over the length of planning horizons k commences (Line 3). When the algorithm comes to increment k without having found a plan, then $I \nvDash \mathcal{L}_k$, and

Algorithm 4: FOND-PDR

Input: A FOND planning problem $\langle X, A, I, G\rangle$
Output: *True* if a policy exists for I, *False* otherwise

```
1  if I ⊨ L_0 then return True
2  L_0 ← G, ∀i > 0 : L_i ← ∅
3  for k ∈ {1, ..., CT} do
4      Reset then forward-push the layer clauses
5      if I ⊭ L_k and L_k ≡ L_{k−1} then return False
6      Iteratively push the layer clauses
7      Q.add_unlocked(⟨I, k⟩)
8      while ¬Q.empty() do
9          ⟨s, i⟩, UA ← Q.get_computing_UA()
10         if ⟨s, i⟩ is progressable via a, a ∉ UA then
11             Q.add_unlocked(⟨s, i⟩)
12             for s' ∈ succ(s, a) do
13                 j ← the lowest j such that s' ⊨ L_j
14                 Q.add(⟨s', j⟩)
15             if policy generator has a policy for I then
16                 return True
17         else
18             if |UA| = 0 then
19                 r ← generate a reason from s
20                 for j ∈ {0, ..., i} do L_j ← L_j ∧ r
21                 if i < k then
22                     Q.add_unlocked(⟨s, i + 1⟩)
23                 Q.inform_of_reason(r, i)
24             else
25                 Q.add_locked(⟨s, i⟩)
26 return False
```

therefore there is no policy π such that $\lceil \pi \rceil_I \leq k$ (further details of this are provided in Theorem 2). We increment k up to a completeness threshold. In the FOND context, this is a value such that if a policy π does not exist such that $\lceil \pi \rceil_I \leq CT$, then no policy exists at all. We here use the trivial completeness threshold of $(2^{|X|} - 1)$ – i.e. if no policy exists even if it can encounter all possible states then no policy exists at all.

At the beginning of each k-iteration we test for the non-existence of a policy for I (Lines 4 and 5). This is done via forward-pushing. The layer formula is first reset to how it is on Line 2 – i.e. – setting $\mathcal{L}_0$ to be the goal, and all other layers to be the vacuously satisfied empty formula. Then an attempt is made to forward-push all the recently cleared reasons to $\mathcal{L}_1$. Then, when $k > 1$, all the reasons that were successfully added to $\mathcal{L}_1$ are forward-pushed to $\mathcal{L}_2$. This process is repeated, pushing out to $\mathcal{L}_k$. Then, if $\mathcal{L}_k \equiv \mathcal{L}_{k-1}$ and $I \not\models \mathcal{L}_k$ the algorithm returns *False* (Line 5). A proof of why the algorithm can safely declare no policy exists is provided in Theorem 2.

The layer formula needs to be forward-pushed in order to perform the policy non-existence check. After the check is performed however, we can further push clauses to increase efficiency. The same iterative pushing procedure is performed as before, but reasons can now be pushed, instead of forward-pushed. This procedure is repeated until it reaches a fixpoint where no reasons can be pushed any further (Line 6). Note, this step is included for efficiency reasons only, and is not logically required.

At this point, an obligation for the initial state $\langle I, k \rangle$ is added to Q (Line 7), and then a loop starting on Line 8 is entered which is broken once the queue is empty. This indicates that work will continue at the current value of k until there are no obligations on Q to evaluate. The loop starts by taking an obligation $\langle s, i \rangle$, along with its unauthorized actions UA off the queue (Line 9). A test is made to see if the obligation can be progressed by an authorized action (Line 10). If it can, then the original obligation, as well as obligations generated from the outcomes are added to the queue (Lines 11 - 14). At this point, the policy generator may have enough information to declare a policy exists for the initial state, if it does, then *True* is returned (Line 16). We note that if the actual policy is wanted for use by a human operator of FOND-PDR, it can be provided by the policy generator in Algorithm 5. If the obligation cannot be progressed, then what happens next is dependent on if there are unauthorized actions (Line 18). If there are no unauthorized actions, a reason is generated and used to constrain the layer formula (Lines 19 and 20). Additionally, this reason is given to the queue which uses it for its internal workings – further details are provided in Section 6.3 (Line 23). If the obligation's index i is

less than k, then the original obligation is modified, and given back to the queue with an incremented index – as it is now inconsistent with the layer formula at its original index (Line 22). If there are unauthorized actions, then the obligation is given back to the queue, noting that it could not be progressed (Line 25). This obligation is stored in the queue until there are no more unauthorized actions for it, then an attempt is made to progress it again.

After Line 25 the k-iteration ends. The layer formula derived during this iteration cannot be carried over to the next iteration as is, because when k changes, the $\mathcal{L}_k$ formula referred to in the progressability test also changes.

6.3 The Queue

We now explain the queue in more detail. There is only ever one queue, but the queue internally maintains two sets of obligations, *locked*, and *unlocked*. We say the queue is empty if there are no unlocked obligations. Every obligation in the queue has an accompanying set of unauthorized actions. These are all the actions that have progressed, and still can progress that obligation. Recall that as the successor and k'th layers are constrained over time, whether or not an obligation can be progressed by an action can become increasingly constrained as the algorithm progresses. Whenever an obligation is de-queued (Line 9), an obligation from the unlocked set with the lowest layer index is returned, along with its corresponding set of unauthorized actions.

Obligations can be added to either the unlocked or locked set. Adding to the unlocked set signifies the obligation is ready to be progressed. We add to the locked set only when all the actions that could progress an obligation are unauthorized (Line 25). In this case, we wait until all the unauthorized actions become authorized, and move this obligation to the unlocked set. Whenever the policy generator discovers a policy for a state, that state is forever removed from the queue and prevented from being added to the queue forevermore. This prevents FOND-PDR searching for policies for states which already have one. When the queue is *informed of a reason* (Line 23), it uses this to update what actions will be authorized, as well as modify obligations in the queue. When a reason r at layer j is found, each obligation $\langle s, i\rangle$ such that $s \vDash r$ and $i \leq j$, is modified to be $\langle s, j+1\rangle$, unless $j = k$, in which case the obligation is removed – matching the queue trimming process of classical PDR. This is to enforce that all states in the queue are at the lowest layer they are consistent with – i.e., we maintain an invariant (stronger but similar to the invariant of Lemma 4 of classic PDR), that $\langle s, i\rangle$ being in the queue means $s \vDash \mathcal{L}_i$ and $s \nvDash \mathcal{L}_{i-1}$.

Algorithm 5: Compute Solved States

Input: Discovered hypergraph $\mathcal{H} = (V, E)$, goal states V^G
Output: States with policies in (V, E)

```
1 loop
2     V^⊥ ← {v | v ∈ V, ∄v^G ∈ V^G s.t. v ⇝_H v^G}
3     if |V^⊥| = 0 then return V
4     else Remove all V^⊥ from H along with any hyperarcs adjacent an element
       in V^⊥
```

It is impossible for all queued obligations to become locked. When there is a locked obligation, it will become unlocked when all of its unauthorized actions become authorized again. Each unauthorized action is dependent on a set of corresponding successor obligations. These successor obligations may themselves be locked. This can form a large network of obligations being locked and dependent on other obligations. However if there are locked obligations, there will always be at least one unlocked obligation. Proceeding with an argument by contradiction, suppose all obligations in the queue are locked. Then, there must be a (or multiple) obligations with the lowest index of all locked obligations. In order for an obligation to be locked, the corresponding state must have unauthorized actions. Each of these actions must be able to progress the obligation. In order to progress the obligation, at least one successor state must be at a lower layer, which is impossible in the context of our premise as this obligation has the lowest layer index of any obligation on the queue – i.e., only if a policy had been found for the state corresponding to the obligation would something like this be possible, however of course we do not have obligations on the queue where the corresponding states have known policies.

Here we describe our policy generator, which runs according to Algorithm 5. This algorithm takes the known state-space hypergraph (V, E), as well as known goal states V^G. The algorithm operates by modifying a copy of this graph. When we remove a state/vertex from this graph, we also remove all hyperarcs the state participates in. We define a *sink* state to be a state which has no path to the goal in the currently explored state-space graph. As such, we can safely say no policy over discovered states exists for these states. If a state-space graph has no sink states, then every state has the possibility of getting to the goal. As such, for all states in this graph, a policy is simply to perform a random action in the state-space graph, as eventually—assuming fairness—a goal state will be reached – i.e., as all states have a chance to get to the goal, by repeatedly transitioning one "can't go wrong".

Algorithm 5 operates in this fashion, initially assuming all discovered states have

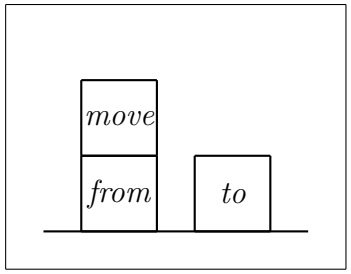
a policy, then repeatedly removing sink states until only states and actions that form parts of policies remain. The states which can reach a goal state are found, and all other states can then safely be determined to be sink states (Line 2). We then remove these states from (V, E), and iterate this process (Line 4). If no sink states are found (Line 3), then all remaining states have a path to the goal, so repeated application of any remaining action will eventually lead to the goal (under the fairness assumption), so all remaining states have a policy. Note in the practical implementation, an online version is used, only running periodically and considering the components which have changed since it was last run.

This algorithm is similar to the Strong-Cyclic-Plan algorithm presented in Ghallab et al. (2004b). The main difference is that this algorithm considers the entire state-space hypergraph, and uses the trimming of sink states as the key search mechanism to find a policy.

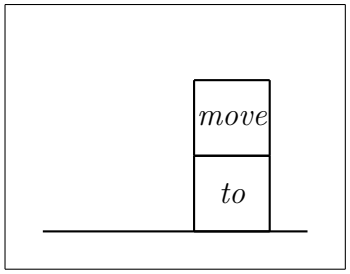
An example of the operation of FOND-PDR is provided in Example 9.

Example 9 (FOND-PDR **Execution on an Example Problem)**

We demonstrate the execution of FOND-PDR *on the Blocksworld problem presented in Example 2. In this domain, there are three blocks, to, from, and move. Each block can be on the table, held by an operator, or on another block. The initial state has move on from, and to and from on the table. The goal condition is that move is on to. We show this graphically in here:*[1]



Initial State



Goal Condition

We trace FOND-PDR *in a similar fashion to Example 5. When showing the state of the layers/queue, we have separate rows for the locked and unlocked components of the queue. For each obligation in the queue, we also show the associated banned actions.*

Initially, the obligation $\langle I, 1\rangle$ *is added to the queue unlocked (Line 7):*

Index	0	$k = 1$
$\mathcal{L}$	$\neg on(move, to)$	
$Q - Unlocked$		$onTable(to) \wedge onTable(from) \wedge on(move, from)$
$Q - Locked$		

This initial obligation is taken off the queue, with no banned actions. An attempt is made to progress it which fails, because no applicable action has an outcome satisfying $\mathcal{L}_0$*. A reason* $on(move, from)$ *is found and used to constrain the layer formula.*

[1] This is the same as Figure 1.2, repeated for convenience.

Index	0	$k = 1$
$\mathcal{L}$	$\neg on(move, to)$ $on(move, from)$	$on(move, from)$
$Q - Unlocked$		
$Q - Locked$		

The k-iteration ends, and k is incremented to 2*. The layer formula is reset, and an attempt to forward push the clauses returns the layer formula to how they were before the end of the previous k-iteration (Line 4). The convergence check and clause pushing has no effect (Lines 5 and 6). The initial obligation* $\langle I, 2\rangle$ *is added to the queue unlocked (Line 7):*

Index	0	1	$k = 2$
$\mathcal{L}$	$\neg on(move, to)$ $on(move, from)$	$on(move, from)$	
$Q - Unlocked$			$onTable(to) \wedge onTable(from) \wedge on(move, from)$
$Q - Locked$			

This initial obligation is taken off the queue to be progressed. Attempting to pick move up from $from$ *results in two outcomes:*

- *One outcome corresponds to the block being successfully picked up:* $onTable(to) \wedge onTable(from) \wedge holding(move)$,
- *The other corresponds to the move block being dropped on the table:* $onTable(to) \wedge onTable(from) \wedge onTable(move)$.

Both these states are consistent with layer 1, and corresponding obligations are added to the queue. Additionally, this action is banned for the originating obligation:

Index	0	1	$k = 2$
$\mathcal{L}$	$\neg on(move, to)$ $on(move, from)$	$on(move, from)$	
$Q - Unlocked$		$onTable(to) \wedge onTable(from) \wedge onTable(move)$ $onTable(to) \wedge onTable(from) \wedge holding(move)$	$onTable(to) \wedge onTable(from) \wedge on(move, from)$ $-BA = \{pickUp(move, from)\}$
$Q - Locked$			

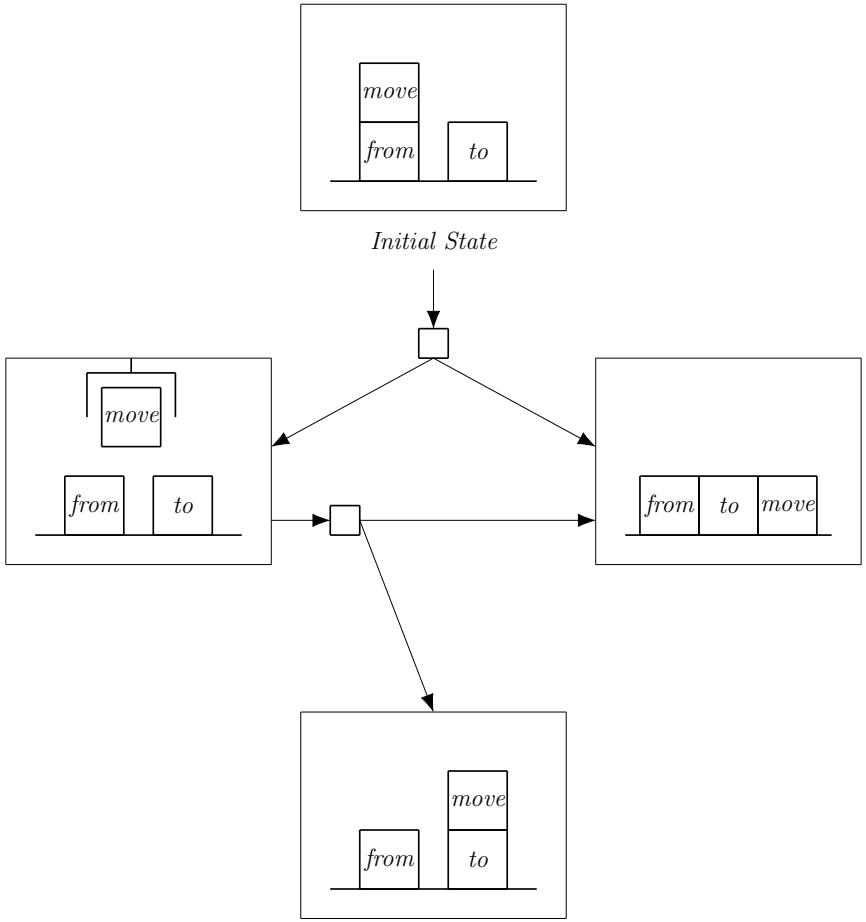
An unlocked obligation with the lowest layer index is taken off the queue to be progressed, here $\langle onTable(to) \wedge onTable(from) \wedge holding(move), 1\rangle$ *is chosen. Attempting to put move down on to results in two outcomes:*

- *One outcome corresponds to this action being successful:* $onTable(to) \wedge onTable(from) \wedge on(move, to)$*, which is consistent with layer* 0 *– i.e. – a goal state.*
- *The other outcome is dropping the block:* $onTable(to) \wedge onTable(from) \wedge onTable(move)$ *which is consistent with layer* 1*.*

The goal state is not added to the queue because it is a goal state, and the other is already on the queue. This action is then added as a banned action to the originating obligation:

Index	0	1	$k = 2$
$\mathcal{L}$	$\neg on(move, to)$ $on(move, from)$	$on(move, from)$	
$Q - Unlocked$		$onTable(to) \wedge onTable(from) \wedge onTable(move)$ $onTable(to) \wedge onTable(from) \wedge holding(move)$ $-BA = \{putDown(move, to)\}$	$onTable(to) \wedge onTable(from) \wedge on(move, from)$ $-BA = \{pickUp(move, from)\}$
$Q - Locked$			

At this point, the policy generator's state-space graph can be represented graphically, where outgoing arrows to small boxes represent actions, and arrows from those boxes to other states represents outcomes of those actions:



Currently, $onTable(to) \wedge onTable(from) \wedge onTable(move)$is a sink state, so no policy is known. $\langle onTable(to) \wedge onTable(from) \wedge$

$holding(move), 1\rangle$ is taken off the queue again to be progressed. However $putDown(move, to)$ is banned, and no other actions are applicable, so this obligation is added to the locked queue:

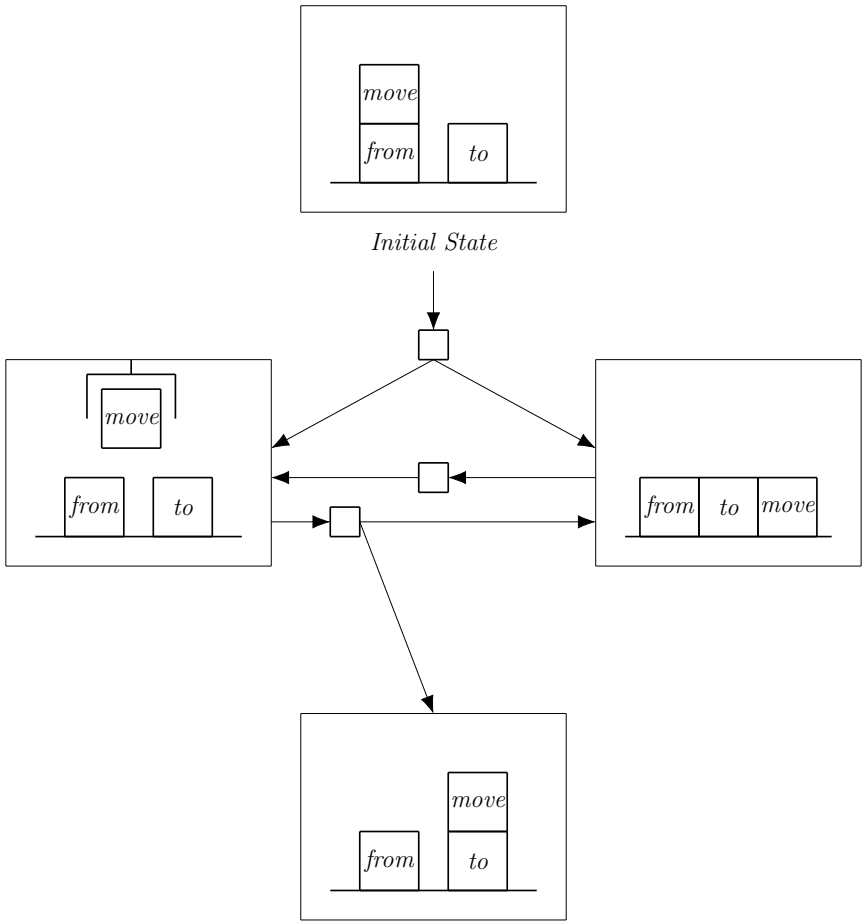
Index	0	1	$k = 2$
$\mathcal{L}$	$\neg on(move, to)$ $on(move, from)$	$on(move, from)$	
$Q - Unlocked$		$onTable(to) \wedge onTable(from) \wedge onTable(move)$	$onTable(to) \wedge onTable(from) \wedge on(move, from)$ $-BA = \{pickUp(move, from)\}$
$Q - Locked$		$onTable(to) \wedge onTable(from) \wedge holding(move)$ $-BA = \{putDown(move, to)\}$	

$\langle onTable(to) \wedge onTable(from) \wedge onTable(move), 1\rangle$ is then taken off the queue to be progressed. This fails, so a reason $onTable(move)$ is generated and used to constrain the layer information. Additionally, this obligation is rescheduled to layer 2.

Index	0	1	$k = 2$
$\mathcal{L}$	$\neg on(move, to)$ $on(move, from)$ $onTable(move)$	$on(move, from)$ $onTable(move)$	
$Q - Unlocked$			$onTable(to) \wedge onTable(from) \wedge on(move, from)$ $-BA = \{pickUp(move, from)\}$ $onTable(to) \wedge onTable(from) \wedge onTable(move)$
$Q - Locked$		$onTable(to) \wedge onTable(from) \wedge holding(move)$ $-BA = \{putDown(move, to)\}$	

This rescheduled obligation $\langle onTable(to) \wedge onTable(from) \wedge onTable(move), 2\rangle$ is then taken off the queue to be progressed. The action of picking move up from the table has the sole outcome: $onTable(to) \wedge onTable(from) \wedge holding(move)$, and this action is executed.

At this point, the policy generators state-space graph can be represented graphically as:



There exists a policy for the initial state, so FOND-PDR *terminates returning* True.

6.4 Progressing Obligations via SAT

The test for progressability is done via SAT, we here describe the encoding. To denote a proposition $p \in X$ is *True* or an action $a \in A$ is executed at the originating time-step we simply write these variables as is. We introduce the idea of *time-slices* – similar but distinct to the time-steps used in classical PDR – to refer to propositions in each of the possible alternate futures arising from the execution of an action. In other words, for each distinct action outcome we have a distinct possible future state. In this chapter, for a formula F, we write F^i to denote F in the successor time-slice corresponding to the i'th possible alternate future – i.e. – corresponding to the i'th outcome being the outcome "chosen" by nature. We call this collection of successor time-slices the *all-futures* time-slices. The SAT instance describing one execution of a non-deterministic action is created with enough possible futures so that every action can be represented – i.e. – the number of futures in the encoding is equal to the maximum number of outcomes of any action. For the actions with less than this many outcomes, the variables corresponding to alternate futures that the action does not address, are ignored.

As part of obligation progression, at least one outcome has to be constrained by the successive layer formula. We call this outcome the *progress outcome*. We write F^{PO} to denote that F exists within the successor time-slice corresponding to this chosen outcome. For an action a and an outcome $o \in \mathcal{O}(a)$, we write $PO(o)$ to denote that this outcome is the progress outcome. When a SAT instance is satisfiable, then the successor states along with which action was executed can be extracted from the satisfying assignment (Line 10). With this notation, we now describe the encoding in Schemas 6-15.

Schema 6 (Time-Slices Constrained by Layer Clauses)
First of all, the successor and k layer formula constrain the progress outcome time-slice and the all-outcomes time-slices respectively. For the obligation $\langle s, i \rangle$, and where n is the maximum number of outcomes of any action, we have:

$$\mathcal{L}_{i-1}^{PO} \wedge \bigwedge_{j=1}^{n} \mathcal{L}_k^j$$

When forward-pushing, there are no restriction placed on the all-outcomes time-slices, so instead we just have:

$$\mathcal{L}_{i-1}^{PO}$$

For ease of presentation, all other formulae are identical. ■

Schema 7 (Action Implies Precondition)
For each action, we impose that it can only be executed if its preconditions are met.

For each action $a \in A$, for each literal in its precondition $p \in \text{pre}(a)$ we have:

$$a \Rightarrow p$$

■

Schema 8 (Action Implies Effects in All-Futures Effects)
If an action is executed, then the all-futures successor states must be consistent with their relevant effects. For each action $a \in A$, for each outcome $o \in \mathcal{O}(a)$, and its corresponding outcome index i, for each literal in its corresponding effect $e \in \text{eff}(o)$ we have:

$$a \Rightarrow e^i$$

■

Schema 9 (Progress Outcome Implies Effect)
We constrain the progress outcome time-slice to be consistent with the effects of the chosen progress outcome. For each action $a \in A$, for each outcome $o \in \mathcal{O}(a)$, for each literal in its corresponding effect $e \in \text{eff}(o)$ we have:

$$PO(o) \Rightarrow e^{PO}$$

■

Schema 10 (Progress Outcome Frame Axioms)
We include frame axioms *for the progress outcome time-slice. These are formula encoding that a proposition's polarity does not change unless made so by an outcome effect. For each proposition $p \in X$ we have:*

$$p^{PO} \Rightarrow p \vee (PO(o_1) \vee \cdots \vee PO(o_m))$$

Where $o_1 ... o_m$ are all the outcomes which have p in their effects. Similarly we also have:

$$\neg p^{PO} \Rightarrow \neg p \vee (PO(o_1) \vee \cdots \vee PO(o_m))$$

Where $o_1 ... o_m$ are all the outcomes which have $\neg p$ in their effects. ■

Schema 11 (All-Futures Frame Axioms)
We include frame axioms for the all-futures time-slices. For each outcome index i, for each proposition $p \in X$ we have:

$$p^i \Rightarrow p \vee (a_1 \vee \cdots \vee a_m) \vee (a_1 \vee \cdots \vee a_n)$$

Where $a_1, \ldots, a_m$ are all the actions which have p in their i'th outcome effects – i.e. – every action a such that p is in $\text{eff}(o_i)$ where $\mathcal{O}(a) = \{o_1, .., o_k\}$. Also, $a_1, \ldots, a_n$ are all the actions which do not have an i'th outcome – i.e. – if an action is chosen which does not "use" all the outcome possibilities, do not constrain those propositions.

Similarly for the negation, for each outcome index i, for each proposition $p \in X$ we have:

$$\neg p^i \Rightarrow \neg p \vee (a_1 \vee \cdots \vee a_m) \vee (a_1 \vee \cdots \vee a_n)$$

Where $a_1, \ldots, a_m$ are all the actions where have $\neg p$ in their i'th outcome effects, and $a_1, \ldots, a_n$ are all the actions which do not have an i'th outcome. ■

Schema 12 (Action Implies at Least One Corresponding Progress Outcome)
For each action $a \in A$, where $\mathcal{O}(a) = \{o_1, .., o_k\}$ we have:

$$a \Rightarrow (PO(o_1) \vee \cdots \vee PO(o_k))$$

■

Schema 13 (Progress Outcome Implies Corresponding Action)
For each action $a \in A$, for each outcome $o \in \mathcal{O}(a)$, we have:

$$PO(o) \Rightarrow a$$

■

Schema 14 (Only One Progress Outcome Per Action)
For each action $a \in A$, where $\mathcal{O}(a) = \{o_1, .., o_k\}$ we have:

$$\bigwedge_{i=1}^{k} \bigwedge_{j=i+1}^{k} (\neg PO(o_i) \vee \neg PO(o_j))$$

Where k is the number of outcomes of a. ■

Schema 15 (Only One Action)
We add clauses so that only one action can be executed. Instead of using the quadratic encoding, as was used in the at-most-one encoding of Schema 14, we utilize auxiliary variables to reduce the number of clauses. The method proceeds as follows:

1. *First, if there are no actions, then the problem has no solution unless $I \vDash G$.*
2. *If there is only a single action, enforce this action and break.*
3. *Initialize a queue to contain all action variables. When modifying this queue, variables are taken off from one end, and added to the other end.*
4. *If there are only two elements in the queue (a and b), add the clause:*

   $$\neg a \vee \neg b$$

   Then break.
5. *If there are more than 2 elements in the queue, pop two elements (a and b), and create an auxilliary variable c. Then add the clauses:*

   $$\neg a \vee \neg b, a \Rightarrow c, b \Rightarrow c$$

Then add c to the queue. Go to step 4.

As in the ∀-step encoding for classical planning, we also add binary mutex invariant clauses to all time-slices to accelerate the search. ■

6.5 FOND-PDR is Sound and Complete

We show that FOND-PDR is sound and complete, finding strong cyclic solutions where they exist, and otherwise proving none exist. The proof of this is broken into three parts, showing:

- FOND-PDR terminates either producing a policy or declaring none exists,
- Any policy produced is valid.
- If FOND-PDR declares no policy exists, then no policy exists.

FOND-PDR only declares a policy exists when the policy generator declares that one exist, so we here show termination and the declaration of the non-existence of a policy.

6.5.1 FOND-PDR Terminates

There are finitely many states, each having a finite number of actions executable in them. There are correspondingly a finite number of reasons. As such, there can only be a finite number of iterations of the inner $\neg Q$.empty() loop, as each iteration either progresses a new action, or discovers a new reason, and these can only happen a finite number of times. The outer k-iteration loop has a maximum number of iterations, so FOND-PDR will eventually terminate.

6.5.2 Declaring No Policy Exists

We first present Theorem 2 which is later used in a larger proof.

Theorem 2 (For every state s, Algorithm 4 maintains the invariant that if $s \not\models \mathcal{L}_i$ there is no policy π such that $\lceil \pi \rceil_s \leq i$)
Proof. Initially, $\mathcal{L}_0 \equiv G$, so every state s such that $s \not\models \mathcal{L}_0$ is not a goal state. All other layers are initially satisfied by any state.

Layer formulae are only ever modified to reset them back to this starting configuration, or by the addition of valid reasons, either through pushing/forward-pushing (Lines 4 and 6), or directly on Line 20. When a reason is added to a layer formula, it is only because all states with which it is consistent cannot be progressed. As such, for all actions, there is at least one successor state s' which does not satisfy

$\mathcal{L}_{i-1}$ – i.e. all actions have at least one successor state s', where there does not exist a policy π such that $\lceil\pi\rceil_{s'} \leq (i-1)$. Since progressing to that successor state would take one step, there is no policy π, for any state s consistent with r, such that $\lceil\pi\rceil_s \leq (i-1)+1 = i$. As such constraining $\mathcal{L}_i$ with reason r does not violate this invariant. ■

On Line 4, layer formula are forward-pushed. On Line 5, if this forward-pushing results in $\mathcal{L}_k \equiv \mathcal{L}_{k-1}$ this process can be extended to forward-push this same layer formula to $\mathcal{L}_{k+1}$. This is guaranteed to be successful, as the criteria for the formula being forward-pushed to $\mathcal{L}_k$ (which is $\mathcal{L}_{k-1}$), is now the same as the criteria for the formula being pushed to $\mathcal{L}_{k+1}$ (which is $\mathcal{L}_k$). This same argument can be applied repeatedly, pushing this layer formula out arbitrarily far. When $I \not\models \mathcal{L}_k$ (Line 5), then as per Theorem 2, for any policy π, $\lceil\pi\rceil_I$ can be shown to be arbitrary large. Since there are only finitely many states, then there is no policy for I.

Otherwise, FOND-PDR returns *False* on Line 26. In this case the obligation $\langle I, CT\rangle$ had been added to the queue. Since the inner loop at Line 8 has exited then a reason r such that $s \models r$ has been used to constrain $\mathcal{L}_{CT}$. Then identically to the argument above, since $I \not\models \mathcal{L}_{CT}$, by Theorem 2 there is no policy for I.

6.6 Experimental Evaluation

We evaluate the performance of FOND-PDR against three other leading FOND solvers, PR2 (Muise et al. 2024), Paladinus (Pereira et al. 2022) and FOND-SAT (Geffner and Geffner 2018). We implement FOND-PDR in `C++`, using a modified version of Madagascar as a parser and preprocessor, and its invariant computation extended to cover non-deterministic effects. As in other PDR variants, we use Lingeling as the SAT solver. We use the FOND benchmark set provided by (Geffner and Geffner 2018).[2]

Additionally, we provide a domain we call *Escher-Blocksworld* to highlight the difference FOND-PDR has to the other solvers. All instances of this domain have no policies. This domain behaves similarly to the existing Blocksworld domain, where there are a set of blocks, blocks can either be on a table or on another block. The initial state has all blocks on the table, but the goal condition requires blocks to be stacked upon each other in a circular nature. For n blocks, for all $0 \leq i < n$ we require block $i+1$ to be on block i, but then also require block 0 to be on block n, which is impossible. The set of actions provided in the Blocksworld domain actually

[2]Downloadable at https://github.com/tomsons22/FOND-SAT.

allows for this to occur due to the stacking/unstacking of *towers* actions, so these have been removed.

All experiments ran on a host with 93 GB of memory with an Intel(R) Xeon(R) Gold 6134 CPU. All problem instances were run with a 30-minute (1800s) timeout.

Table 6.1 shows coverage data for the evaluated planners. For each planner, the number of instances solved along with the median time is shown. The best result for each domain is shown in bold. Note FOND-SAT cannot solve the instances where no policy exists. On the Escher-Blocksworld domain, FOND-PDR has the best performance out of all the solvers tested, solving all instances almost instantly. Since the goal state is impossible to reach with any action, during the forward-pushing phase (Line 4), every reason in $\mathcal{L}_0$ can be pushed to $\mathcal{L}_1$. Then $I \not\models \mathcal{L}_1$, so *False* can be immediately returned.

6.7 Conclusion

We develop the first algorithm based on PDR for solving the strong cyclic FOND planning problem. We developed a new invariant condition for PDR that recognizes that actions here are non-deterministic, having multiple possible outcomes, any one of which nature chooses at each execution. We empirically demonstrate that planning in an AND/OR graph using states discovered by FOND-PDR obligation processing is competitive with current state-of-the-art FOND algorithms, giving best performance in some benchmarks.

Compared to three other leading FOND solvers, FOND-PDR has the second best overall coverage, the equal best coverage in four domains, and the unique best coverage in three domains. A familiar strength of PDR is in being able to detect when goal-achieving paths do not exist, and we are able to exploit and demonstrate that strength of FOND-PDR.

		Number of Instances Solved (Median Time)			
Domain	Instances	PR2	FOND-SAT	Paladinus	FOND-PDR
Acrobatics	8	**8** (0.7)	3 (N/A)	**8** (0.6)	6 (4.1)
Beam-Walk	11	**11** (0.8)	2 (N/A)	9 (1.3)	5 (N/A)
Blocksworld-New	50	**49** (43.4)	7 (N/A)	13 (N/A)	5 (N/A)
Chain-Of-Rooms	10	**10** (0.4)	0 (N/A)	**10** (0.7)	**10** (84.4)
Doors	15	**15** (0.3)	9 (398.1)	13 (2.0)	**15** (0.3)
Earth-Observation	40	**40** (0.3)	5 (N/A)	18 (N/A)	17 (N/A)
Elevators	15	**15** (0.3)	7 (N/A)	9 (11.8)	14 (1.1)
Faults-New	190	**190** (16.0)	6 (N/A)	21 (N/A)	22 (N/A)
First-Responders-New	88	**87** (37.1)	4 (N/A)	1 (N/A)	8 (N/A)
Forest-New	100	**91** (1.2)	10 (N/A)	10 (N/A)	0 (N/A)
Islands	60	**60** (0.4)	56 (29.1)	**60** (0.6)	**60** (0.9)
Miner	51	**51** (0.7)	44 (489.5)	**51** (2.3)	**51** (14.7)
Tidyup-MDP	10	**10** (0.7)	0 (N/A)	1 (N/A)	0 (N/A)
Tireworld	12	**12** (0.3)	**12** (2.2)	**12** (0.4)	**12** (0.2)
Tireworld-Spiky	11	**11** (0.3)	1 (N/A)	10 (24.9)	4 (N/A)
Tireworld-Truck	74	**74** (0.4)	56 (261.6)	30 (N/A)	**74** (0.4)
Triangle-Tireworld	40	**40** (39.9)	2 (N/A)	3 (N/A)	4 (N/A)
Zenotravel	15	**15** (1.8)	5 (N/A)	5 (N/A)	9 (202.7)
Escher-Blocksworld	20	0 (N/A)	0 (N/A)	2 (N/A)	**20** (**0.4**)
Total	820	**785**	229	284	316

Table 6.1: Number of instances solved and median times for different planners. The best performance for each domain is in bold.

Chapter 7

Conclusion

7.1 Summary of Contributions

PDR is a powerful sound and complete classical planning search paradigm. Throughout this thesis we have explored many facets of its operation in order to expand and accelerate them.

SAT solvers have been shown to be a powerful tool for solving planning problems. PDR involves many calls to a SAT solver, however each of these calls consist of small SAT instances corresponding to a small single step planning problem. In Chapter 3 we presented two solvers which offload work away from PDR's internal logic to the highly optimized SAT solvers used to evaluate the small SAT instances. When taken to its extreme, this approach would effectively revert back into direct SAT planning. The extent to which the work is offloaded is parameterisable, and we explored how different parameter choices affect performance.

Since its inception, PDR has been identified as being amenable to parallelism. In Chapter 4 we present a new parallel version of PDR PS-PDR which processes obligations independently in parallel. There exists parallel variants of PDR which instead operate by having a collection of PDR processes all sharing learnt nogoods amongst each other. These variants cannot be directly compared against PS-PDR as they are not designed for planning problems. Because of this we created a solver designed to reimplement these existing solvers for planning so they can be compared against our new solutions.

Instead of treating the planning problem as a monolythic transition system, in Chapter 5 we present a solver which analyses the structure of the problem in order to decompose it into a collection of subproblems, which can be solved independently in parallel.

PDR has been an effective solver for classical planning problems, able to reason about complicated transition systems without unrolling, and through reason finding make statements about many related states all at once. These attributes are useful

beyond classical planning. In Chapter 6 we adapt PDR to work in the FOND setting, where the state-space can often get unwieldy and complicated. We developed many adaptations for PDR for it to perform in a FOND environment, while still keeping to the design principles behind classical PDR.

All presented solvers have been implemented with available codebases, and experimentally evaluated against relevant comparison solvers. Appendix A presents a table of all solvers introduced in this thesis.

7.2 Future Work

We expect the following future research threads will be fruitful:

- Many of the research threads presented in this paper are orthogonal and could be combined to exacerbate their benefits – e.g. One could imagine a variant of PD-PDR, where each subproblem is solved with PS-PDR with modified SAT calls as per PDR-M/PDR-IL. In the most extreme case, all four main (chapters 3 - 6) PDR alterations could be combined. This would result in a FOND solver, with multiple planning steps per SAT call, running in parallel on decomposed problems. Doing this gracefully however would not be trivial, most notably:
  - The decomposition of planning problems as used by PD-PDR would need to be adapted to a FOND setting.
  - Having multiple planning steps in a single SAT call (as in PDR-M and PDR-IL) relies on a single state being the successor of the considered state, so that actions can be chained. In a FOND setting, there would be an exponential explosion of states to be considered together, as each action produces multiple outcomes, and each of those outcomes produce multiple outcomes.

  In this thesis we do not consider the various combinations this could take, presenting an ablation study on these techniques individually. Future works should consider combining our advances.
- PD-PDR breaks a problem into a collection of classical planning problems. Here we use serial PDR to solve each of these problems, to compare this with other PDR-based methods. However it may be interesting to use different complete solvers in its place.
- We have adapted a reason minimisation strategy from the literature for PDR. In our experience there remains a clear opportunity to investigate heuristics

for reason minimisation in PDR for planning. We have observed that hand crafted strategies can in specific circumstances make a substantial difference in the performance of planning overall.

- Obligations are taken from the queue in the order of layer index (lowest index first), and action selection is at the whim of the SAT solver. Exploration of the use of heuristics for improved selection of obligations to progress and actions to progress them, could be fruitful. Especially in the FOND-PDR case, the heuristics would need to be designed/adapted to work with the queue structure in mind.
- Currently a SAT procedure implementing CDCL is used to solve all SAT instances. SLS is often more effective than CDCL on satisfiable instances, but cannot prove that no solution exists. So when a solution exists/is likely (perhaps according to recent "similar" SAT calls) to exist, a SLS solver could be used in place of the CDCL solver.

  This could be taken even further, replacing the SAT procedure with a custom decision procedure as in (Suda 2014). There are cases where a full SAT procedure is unnecessary – most notably when the successive $\mathcal{L}_{i-1}$ layer only has a few clauses – and it may be effective to use a custom procedure in place of a general use SAT procedure.
- The used SAT procedure is serial. We suspect using a multicore SAT solver could yield some speedups. Indeed, we do not consider different SAT solvers in this work; substituting Lingeling for more advanced SAT solvers as they are developed will surely result in speedups.
- For PDR-M/PDR-IL, different macro lengths can drastically increase/decrease the performance of the solver. The macro length could be adapted online to suit the problem being solved. This could be combined with PS-PDR, with each worker considering a different length macro.
- For the classical planning related sections of this thesis, the $\forall$-step semantics is used. There however exist many other encodings which have been shown to be useful which could be explored instead. In particular, we suppose the $\exists$-step and split encodings could be fruitful.

  Similarly, other improved FOND SAT encodings could be developed to replace the direct encoding we used for FOND-PDR.

Bibliography

Abdulaziz, M. and Berger, D., 2021. Computing plan-length bounds using lengths of longest paths. In *Thirty-Fifth Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence*, 11709–11717. AAAI Press.

Abdulaziz, M.; Norrish, M.; and Gretton, C., 2015. Exploiting symmetries by planning for a descriptive quotient. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, 1479–1486. AAAI Press.

Abdulaziz, M.; Norrish, M.; and Gretton, C., 2018. Formally verified algorithms for upper-bounding state space diameters. *Journal of Automated Reasoning*, 61, 1-4 (2018), 485–520.

Amir, E. and Engelhardt, B., 2003. Factored planning. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 929–935. Morgan Kaufmann.

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

Balyo, T.; Biere, A.; Iser, M.; and Sinz, C., 2016. SAT race 2015. *Artificial Intelligence*, 241 (2016), 45–65.

Bayardo, R. J. and Schrag, R. C., 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, 203–208. AAAI Press.

Biere, A., 2017. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT competition 2017. In *Proceedings of the SAT Competition 2017, Solver and Benchmark Descriptions*, vol. B-2017-1 of *Department of Computer Science Series of Publications B*, 14–15. University of Helsinki.

Biere, A.; Heule, M.; and van Maaren, H., 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications. IOS Press. ISBN 9781607503767.

Bradley, A. R., 2011. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference*, vol. 6538 of *Lecture Notes in Computer Science*, 70–87. Springer.

Brafman, R. I. and Domshlak, C., 2006. Factored planning: How, when, and when not. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*, 809–814. AAAI Press.

Bylander, T., 1994. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69, 1-2 (1994), 165–204.

Chaki, S. and Karimi, D., 2016. Model checking with multi-threaded IC3 portfolios. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference*, vol. 9583 of *Lecture Notes in Computer Science*, 517–535. Springer.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P., 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147, 1–2 (2003), 35–84.

Cimatti, A.; Roveri, M.; and Traverso, P., 1998. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, 875–881. AAAI Press.

Cook, S. A., 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, 151–158. ACM.

Darwiche, A., 2001. Recursive conditioning. *Artificial Intelligence*, 126, 1-2 (2001), 5–41.

Davis, M.; Logemann, G.; and Loveland, D., 1962. A machine program for theorem-proving. *Communications of the ACM*, 5, 7 (Jul. 1962), 394–397. doi: 10.1145/368273.368557.

Eén, N.; Mishchenko, A.; and Brayton, R., 2011. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design*, 125–134. IEEE.

Eén, N. and Sörensson, N., 2003. an extensible sat-solver. In *theory and applications of satisfiability testing, 6th international conference, selected revised papers*, vol. 2919 of *lecture notes in computer science*, 502–518. springer.

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

Geffner, T. and Geffner, H., 2018. Compact policies for non-deterministic fully observable planning as SAT. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling*, 88–96. AAAI Press.

Gerevini, A. and Schubert, L. K., 1998. Inferring state constraints for domain-independent planning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference*, 905–912. AAAI Press / The MIT Press.

Ghallab, M.; Nau, D.; and Traverso, P., 2004a. *Automated Planning: theory and practice*. Elsevier.

Ghallab, M.; Nau, D. S.; and Traverso, P., 2004b. *Automated planning - theory and practice*. Elsevier. ISBN 978-1-55860-856-6.

Gnad, D.; Torralba, Á.; and Fiser, D., 2022. Beyond stars - generalized topologies for decoupled search. In *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling, ICAPS 2022, Singapore (virtual), June 13-24, 2022*, 110–118. AAAI Press. URL `https://ojs.aaai.org/index.php/ICAPS/article/view/19791`.

Gocht, S. and Balyo, T., 2017. Accelerating SAT based planning with incremental SAT solving. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling*, 135–139. AAAI Press.

Hansen, E. A. and Zilberstein, S., 1998. Heuristic search in cyclic AND/OR graphs. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference*, 412–418. AAAI Press / The MIT Press.

Howey, R.; Long, D.; and Fox, M., 2004. VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *16th Institute of Electrical and Electronics Engineers International Conference on Tools with Artificial Intelligence*, 294–301. IEEE Computer Society.

Howson, C., 2005. *Logic with trees: an introduction to symbolic logic*. Routledge.

Huang, J. and Darwiche, A., 2003. A structure-based variable ordering heuristic for SAT. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 1167–1172. Morgan Kaufmann.

Kautz, H.; McAllester, D.; and Selman, B., 1996. Encoding plans in propositional logic. *KR*, 96 (1996), 374–384.

Kautz, H.; Selman, B.; and Hoffmann, J., 2006. SATPLAN: Planning as satisfiability. In *5th international planning competition*, vol. 20, 156.

Kautz, H. A. and Selman, B., 1992. Planning as satisfiability. In *10th European Conference on Artificial Intelligence*, 359–363. John Wiley and Sons.

Kelareva, E.; Buffet, O.; Huang, J.; and Thiébaux, S., 2007. Factored planning using decomposition trees. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 1942–1947.

Knoblock, C. A., 1994. Automatically generating abstractions for planning. *Artificial Intelligence*, 68, 2 (1994), 243–302.

Komuravelli, A.; Gurfinkel, A.; and Chaki, S., 2016. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48 (2016), 175–205.

Marescotti, M.; Gurfinkel, A.; Hyvärinen, A. E. J.; and Sharygina, N., 2017. Designing parallel PDR. In *2017 Formal Methods in Computer Aided Design*, 156–163. IEEE.

Marques-Silva, J. P. and Sakallah, K. A., 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48, 5 (1999), 506–521.

Martelli, A. and Montanari, U., 1978. Optimizing decision trees through heuristically guided search. *Commun. ACM*, 21, 12 (1978), 1025–1039. doi: 10.1145/359657.359664.

Mattmüller, R.; Ortlieb, M.; Helmert, M.; and Bercher, P., 2010. Pattern database heuristics for fully observable nondeterministic planning. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, 105–112. AAAI Press.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D., 1998. PDDL: The planning domain definition language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Muise, C.; McIlraith, S.; and Beck, C., 2012. Improved non-deterministic planning by exploiting state relevance. In *Proceedings of the Twenty-Second In-*

ternational Conference on Automated Planning and Scheduling, 172–180. AAAI Press.

Muise, C.; McIlraith, S. A.; and Beck, J. C., 2024. PRP rebooted: advancing the state of the art in FOND planning. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence*, 20212–20221. AAAI Press.

Nghia Pham, D.; Thornton, J.; Gretton, C.; and Sattar, A., 2008. Combining adaptive and dynamic local search for satisfiability. *Journal on Satisfiability, Boolean Modeling and Computation*, 4, 2-4 (2008), 149–172.

Pereira, R. F.; Pereira, A. G.; Messa, F.; and De Giacomo, G., 2022. Iterative depth-first search for FOND planning. In *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling*, 90–99. AAAI Press.

Puterman, M. L., 1994. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Ramirez, M. and Sardina, S., 2014. Directed fixed-point regression-based planning for non-deterministic domains. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, 235–243. AAAI Press.

Rintanen, J., 1999. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10 (1999), 323–352.

Rintanen, J., 2008. Regression for classical and nondeterministic planning. In *ECAI 18th European Conference on Artificial Intelligence*, vol. 178 of *Frontiers in Artificial Intelligence and Applications*, 568–572. IOS Press.

Rintanen, J., 2012. Planning as Satisfiability: Heuristics. *Artificial intelligence*, 193 (2012), 45–86.

Rintanen, J. and Gretton, C. O., 2013. Computing upper bounds on lengths of transition sequences. (2013), 2365–2372.

Rintanen, J.; Heljanko, K.; and Niemelä, I., 2006. Planning as Satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170, 12-13 (2006), 1031–1080.

Robertson, N. and Seymour, P., 1991. Graph minors. x. obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52, 2 (1991), 153–190.

Robinson, N.; Gretton, C.; Pham, D. N.; and Sattar, A., 2008. A com-

pact and efficient SAT encoding for planning. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, 296–303.

Robinson, N.; Gretton, C.; Pham, D. N.; and Sattar, A., 2009. SAT-based parallel planning using a split representation of actions. In *Nineteenth International Conference on Automated Planning and Scheduling*.

Russell, S. J. and Norvig, P., 2016. *Artificial intelligence: a modern approach*. Pearson.

Selman, B. and Kautz, H. A., 1993. Domain-independent extensions to GSAT: solving large structured satisfiability problems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, 290–295. Morgan Kaufmann.

Selman, B.; Levesque, H. J.; and Mitchell, D. G., 1992. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA, USA, July 12-16, 1992*, 440–446. AAAI Press / The MIT Press.

Silva, J. M. and Sakallah, K. A., 1996. GRASP-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, 220–227. IEEE.

Streeter, M. J. and Smith, S. F., 2007. Using decision procedures efficiently for optimization. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, 312–319. AAAI.

Suda, M., 2014. Property directed reachability for automated planning. *Journal of Artificial Intelligence Research*, 50 (2014), 265–319.

Tarjan, R., 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1, 2 (1972), 146–160.

Turner, H., 2002. Polynomial-length planning spans the polynomial hierarchy. In *tificial Intelligence, European Conference, JELIA 2002*, no. 2424 in Lecture Notes in Computer Science, 111–124. Springer-Verlag.

Vizel, Y.; Weissenbacher, G.; and Malik, S., 2015. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the Institute of Electrical and Electronics Engineers*, 103, 11 (2015), 2021–2035.

Williams, B. C. and Nayak, P. P., 1997. A reactive planner for a model-

based executive. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1178–1185. Morgan Kaufmann.

Zarpas, E., 2005. Benchmarking SAT solvers for bounded model checking. In *International Conference on Theory and Applications of Satisfiability Testing*, 340–354. Springer.

Appendix A

All Introduced Solvers

Solver	Description	Introduction Location	Url
PDR-M	When processing an obligation, instead of transitioning a single step to get to the successor state, multiple states are allowed.	Section 3.1	https://doi.org/10.5281/zenodo.14768741
PDR-IL	Similar to PDR-M, when processing an obligation, instead of transitioning a single step to get to the successor state, multiple states are allowed. However, each of these succesive steps are contrained by a successive layer formula.	Section 3.2	
PS-PDR	A parallel variant of PDR where multiple obligations are processed independantly in parallel.	Chapter 4	https://doi.org/10.5281/zenodo.8020680. Updated version may be available at: https://github.com/ANU-HPC/parallel-pdr
PDR-P	An implementation of an existing PDR parallel technique to the planning setting, to benchmark against.	Section 4.3	
PD-PDR	A method to decompose problems into a collection of subproblems, which are solved in parallel	Chapter 5	
PDR-S	A serial PDR implementation, used to benchmark against.	Section 5.4.1	
FOND-PDR	A variant of PDR designed for FOND problems. mainly modifying the progression semantics, the queue and the stopping criteria to achieve this.	Chapter 6	https://github.com/ANU-HPC/parallel-pdr/tree/nondeterministic

Table A.1: PDR solvers developed for this thesis, their introduction location in the text and where they can be found.