Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning
Main Track

# Property Directed Reachability for Planning Revisited

**Ava Clifton , Charles Gretton**
School of Computing, The Australian National University
ava.clifton.ac@gmail.com, charles.gretton@anu.edu.au

**Abstract**

*Property Directed Reachability* (PDR) is a relatively new SAT-based search paradigm for classical AI planning. Compared to earlier SAT-based paradigms, PDR proceeds without unrolling the system transition function, and therefore without having the underlying procedure reason about potentially computationally expensive multi-step formulae. By maintaining a queue of *obligations*—i.e., a state at a timestep—and knowledge about what is possible at each planning step, PDR iteratively evaluates whether an obligation can be progressed by one step towards the goal. We develop and evaluate two new distributed PDR algorithms for planning, and additionally implement serial and portfolio PDR algorithms for planning. We are the first to consider distributed PDR for planning and the first to consider PDR based on incremental SAT solving in that setting. Our first new algorithm, PS-PDR, evaluates many obligations independently in parallel using a pool of incremental SAT workers. PS-PDR is unique amongst distributed PDR algorithms in centrally maintaining a single queue of obligations, enabling an efficient focused search compared to a PDR portfolio. Our second new algorithm, PD-PDR, sequences subproblems according to the compositional structure of the concrete problem at hand. Subproblems are solved independently in parallel, with a concrete plan obtained by combining subproblem plans. Our experimental evaluation exhibits strong runtime gains for both new algorithms in both satisfiable and unsatisfiable planning benchmarks.

## 1 Introduction

We study the *classical planning* problem. This is the problem of determining whether a finite discrete deterministic transition system can, via a sequence of actions, transition from an initial state to a state satisfying a goal condition. The system is represented succinctly in a formalism such as STRIPS (Fikes and Nilsson 1971) or PDDL (McDermott et al. 1998). A range of solution procedures for this problem proceed by solving a series of Boolean SAT queries, with each query corresponding to a horizon limited version of the planning problem (Kautz and Selman 1992; ; Biere et al. 1999; Rintanen 2004; Streeter and Smith 2007; Rintanen 2012; Gocht and Balyo 2017). The earliest such procedures date back to last century (Kautz and Selman 1992; ; Biere et al. 1999). They proceed by posting a series of SAT queries, each corresponding to a horizon limited version of the planning problem. Recent developments have improved runtime performance using specific tailoring of decision procedures (Rintanen 2012), careful allocation of (distributed) computing resources to queries (Rintanen 2004; Streeter and Smith 2007), and by adopting novel innovations (e.g., incrementality) in SAT solving (Gocht and Balyo 2017). Such approaches face two core difficulties: *(i)* it is difficult to prove that no plan exists, and *(ii)* if the smallest plan is long, they do require a lot of memory because the solution procedure must represent an unrolling of the transition system model over a large number of steps. [1]

PDR is a general sound and complete approach devised to address these core difficulties. First described for a hardware model checking algorithm (Bradley 2011), the approach was subsequently adapted to describe planning algorithms (Suda 2014; Eriksson and Helmert ). PDR operates by iteratively refining reachability information, represented by a formula describing an overapproximation of the set of states that are $N$ steps away from the goal. Queries in PDR, called *obligations*, are 1 step planning problems, asking whether a state satisfying the $N$-step formula can be progressed to a successor state satisfying the $N-1$ step formula. A satisfiable obligation leads to the derivation of a new obligation. An unsatisfiable obligation leads to the derivation of a tighter reachability overapproximation. A satisfiable obligation from the formula 1 step away from the goal, to the goal formula, indicates a plan is found. If no plan exists, a computationally efficient syntactic check on successive reachability formulae determine this.

PDR is recognised as being highly amenable to distributed computing, with existing portfolio algorithms described for model checking (Chaki and Karimi 2016; Marescotti et al. 2017). We develop and evaluate two new complete High-Performance Computing (HPC) PDR algorithms for classical planning. Both use an incremental SAT solver as the underlying decision procedure, with the HPC skeleton based on DAGSTER (Burgess et al. 2022). The first, called PS-PDR, evaluates multiple obligations concurrently in a distributed computing environment using a centrally managed pool of *incremental* SAT solvers. Our second

[1] Given a (tight) completeness threshold the classical *bounded model checking* approach is complete (Baumgartner, Kuehlmann, and Abraham 2002; Kroening et al. 2011; Abdulaziz and Berger 2021).

algorithm, called PD-PDR, 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 well known dependency/causal 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 unsatisfiable as indicated by the unsatisfiability of a subproblem, or the process repeats on a contracted partition until a plan is produced or the concrete planning problem is proved unsatisfiable.

## 2 Background and Notation

We assume familiarity with propositional logic and *incremental* SAT solving (Gocht and Balyo 2017). A *literal* is a formula consisting of a proposition $x$ or its negation $\neg x$. A formula in *Conjunctive Normal Form* (CNF) is a formula consisting of a conjunction of disjunctive clauses. For a formula $F$ and assignment $v$, we write $\texttt{SAT}(\texttt{F})$ to denote that there exists an assignment which satisfies $F$, $\texttt{UNSAT}(\texttt{F})$ if there does not, and $v \vDash F$ to denote that $v$ is a satisfying assignment for $F$ – i.e., $\texttt{SAT}(\texttt{F})$ iff $\exists v.v \vDash F$, and otherwise $\texttt{UNSAT}(\texttt{F})$. 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, $A$ is a set of *actions*, $I$ is the *initial state*, and $G$ is the *goal condition*. 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 $\mathit{pre}(a)$ and an effect formula $\mathit{eff}(a)$. For the sake of a simple exposition, we restrict our attention to precondition and effect formulae that are consistent cubes – i.e., if literal $\ell$ appears in such a cube, then we cannot have $\neg\ell$ also in that cube. An action $a$ is *applicable* in a state $s$ iff $\texttt{SAT}(\texttt{s} \wedge \mathit{pre}(\texttt{a}))$. The state $s' = \mathit{succ}(s, a)$ resulting from executing $a$ at $s$ satisfies: $\texttt{SAT}(\texttt{s}' \wedge \mathit{eff}(\texttt{a}))$ and for every $f \in X$ absent from $\mathit{eff}(a)$ we have that for all $\ell_f \in \{f, \neg f\}$ if $\texttt{SAT}(\texttt{s} \wedge \ell_f)$ then $\texttt{SAT}(\texttt{s}' \wedge \ell_f)$. In the case of a partial state, the successor is not fully determined, and should be taken to be any one total state satisfying the above condition – e.g., for the sake of formalities, we take the lexicographic minimal. An action $a$ *conflicts* with another $a'$ if $\texttt{UNSAT}(\mathit{eff}(\texttt{a}) \wedge \mathit{eff}(\texttt{a}'))$. Those two actions *interfere* if either $\texttt{UNSAT}(\mathit{eff}(\texttt{a}) \wedge \mathit{pre}(\texttt{a}'))$ or $\texttt{UNSAT}(\mathit{eff}(\texttt{a}') \wedge \mathit{pre}(\texttt{a}))$ – i.e., if the preconditions and effects of the two actions are inconsistent. We say that $s_n$ is *instantaneously reachable* from $s_0$, written $\mathit{reachable}^{\forall}(s_0, s_n)$, if there exists a sequence $[s_0, a_0.., s_{n-1}, a_{n-1}, s_n]$ satisfying: *(i)* $\forall i \in \{1, .., n\}, s_i = \mathit{succ}(s_{i-1}, a_{n-1})$, *(ii)* for all pairs of actions $a_i$ and $a_j$ we have that these neither conflict nor interfere, and *(iii)* $\forall i \in \{1, .., n\}, \texttt{SAT}(\texttt{pre}(\texttt{a}_\texttt{i}) \wedge \texttt{s}_0)$. In other words, the set of actions $\{a_0, .., a_{n-1}\}$ can be executed in any order at $s_0$, and the resulting state is $s_n$. Moreover, we can imagine executing that set of actions instantaneously in parallel at $s_0$ to reach $s_n$, as per the $\forall$-step semantics (Rintanen, Heljanko, and Niemelä 2006). We restrict our attention to $\forall$-step semantics in this paper. We call a state $s'$ a *successor* of $s$ if $\mathit{reachable}^{\forall}(s, s')$ holds. A $\forall$-step *execution* of length $n$ between states $s_0$ and $s_n$ is a sequence of states $s_0, s_1, .., s_n$ where successive states $s_i$ and $s_{i+1}$ satisfy $\mathit{reachable}^{\forall}(s_i, s_{i+1})$. A planning problem solution is a $\forall$-step execution from $I$ to a total state $s$ satisfying $s \vDash G$. Such a solution is called a *plan*.

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 facts. 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. Writing $\Sigma(f)$ for the set of all propositions mentioned in a formula $f$, we have: $A{\downarrow}Y = \{a \in A | \Sigma(\mathit{pre}(a)) \cup \Sigma(\mathit{eff}(a)) \subseteq Y\}$. In the case of a cube $s$ and a set $Y$, $s{\downarrow}Y$ is the cube projected to elements in $Y$. For example, $(\neg a \wedge \neg b \wedge c){\downarrow}\{a, c\} = (\neg a \wedge c)$.

Finally, 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 facts $f_1$ 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).

### 2.1 Problem Decomposition for Planning

We develop a new portfolio-style PDR algorithm for planning that first decomposes a concrete problem into a sequence of abstract subproblems that can be solved independently in parallel. The notation and background required for the elicitation of subproblems is described here, and later we will describe our compositional PDR portfolio. 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*, sometimes called a *causal graph*, was first described by (Knoblock 1994; Williams and Nayak 1997). We present the slight variation that is motivated in our setting, and described originally in (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:* (i) *There exists an action $a$ where $x, x' \in \Sigma(\mathrm{eff}(a))$ and $x' \notin X^{\pm}$, or* (ii) *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 $\{v' | \exists v \in \Sigma(G), v \overset{(V,E)}{\rightsquigarrow} v'\}$.*

**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 $v'$ reciprocally reachable from a vertex $v$ in the subgraph is also in the subgraph – i.e., SCCs are maximal."*

**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:* (i) *vertices $\mathbf{P}$ are bijectively labelled by members of a partition of variables $V$, and* (ii) $\mathbf{E}$ *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, Norrish, and Gretton (2018).*

A natural LADG has one vertex for each SCC in the dependency graph, with that set of SCCs providing a suitable partition.

**Example 1** (Analysing a Logistics Domain)**.** *Consider a* Logistics *domain with two* locations*, $L1$ and $L2$, two* packages*, $P1$ and $P2$, and one truck $T$. Objects, namely packages and trucks, can be at locations, and packages can be in trucks. There is one fact for each object-location setting – e.g., at$(P1, L1)$ representing that package $P1$ is at location $L1$. Similarly, there is a variable 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. Figure 1 shows in solid arrows the dependency graph between propositions in this problem, and the dotted ovals and arrows show a corresponding LADG with vertices corresponding to SCCs.*

## 3 Parallel State PDR

We now describe the serial PDR planning algorithm, as well as our first major contribution, namely *Parallel State PDR* (PS-PDR). PDR and PS-PDR are both sound and complete algorithms for solving the planning problem. We describe these algorithms concurrently for reasons of brevity, and also to highlight their similarities. We present pseudocode for PDR/PS-PDR in Algorithm 1. PS-PDR is described given an allocation of $n$ computing cores, which in our context is an 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 $\{1, .., M\}$. We note that with one worker and orchestrator, PS-PDR emulates PDR.

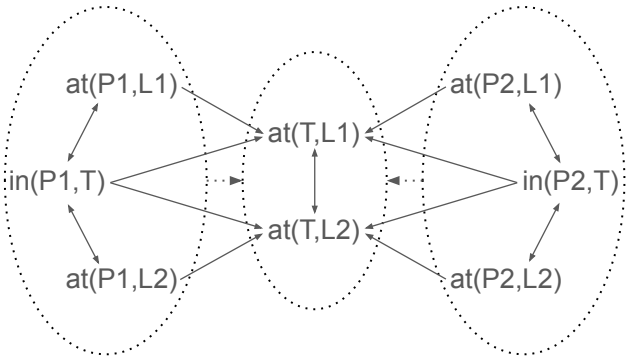



Figure 1: Logistics dependency graph, depicted with a LADG whose vertices, indicated by dashed ovals, correspond to SCCs of that dependency graph.

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 $i$. Formula $\mathcal{L}_i$ satisfies an invariant condition: For every state $s$ such that $s \not\models \mathcal{L}_i$ there is no $\forall$-step execution of length $i$ or less from $s$ to a state satisfying $G$. 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. 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?

PDR proceeds by taking an obligation $\langle s, i\rangle$ from the queue, and querying whether or not there exists a successor state $s'$ 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 state $s$ associated with an obligation at $i$, the layer formula $\mathcal{L}_{i-1}$, and a formula representing the transition function, can be encoded into a single SAT problem which has a solution iff $s$ has a successor $s'$ such that $s' \models \mathcal{L}_{i-1}$. To encode the transition function in SAT we employ the direct $\forall$-step encoding, described by Rintanen(2012).[2] If that formula is satisfiable and $s'$ exists, then the obligation $\langle s', i-1\rangle$ is added to the queue. If no successor state exists, then a *reason* is derived and added to $\mathcal{L}_i$.

A reason, $r$, will be a fragment of $s$ associated with an unsatisfiable obligation at $i$. In other words, $r$ is a partial state consistent with $s$, where $r$ also does not have a successor state satisfying $\mathcal{L}_{i-1}$. Based on the derived reason $r$, a new disjunctive clause is added to $\mathcal{L}_i$, by conjoining the clause $\neg r$. After conjoining, for any state $y$ where $\texttt{SAT}(\texttt{y} \wedge \texttt{r})$ we have $\texttt{UNSAT}(\texttt{y} \wedge \mathcal{L}_\texttt{i})$. 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, partly because incremental solvers provide *used assumptions* (Gocht and Balyo

[2] $\forall$-step is also adopted in PDRPLAN (Suda 2014).

**Algorithm 1:** Distributed PDR

**Input:** A planning problem $\langle X, A, I, G\rangle$, $M$
**Output:** *True* if the problem is solvable, *False* otherwise

1 $Q \leftarrow \{\}, \mathcal{L}_0 \leftarrow G$, for $j > 0 : \mathcal{L}_j \leftarrow$ *True*
2 **if** $I \vDash \mathcal{L}_0$ **then return** True
3 **for** $k \in [1, 2, 3, ...]$ **do**
4  $Q \leftarrow \{\langle I, k\rangle\}$
5  **while** $Q \neq \{\}$, *or* $workers\_waiting() \neq \{1, .., M\}$ **do**
6  **foreach** $w \in workers\_waiting()$ **do**
7  **if** $Q \neq \{\}$ **then**
8  $\langle s, i\rangle \leftarrow$ pop most recently added obligation from $Q$ with minimal $i$.
9  $process\_obligation(w, \mathcal{L}, \langle s, i\rangle)$
10  **foreach** $\langle s', s, i\rangle \in get\_successes()$ **do**
11  **if** $i - 1 = 0$ **then return** True
12  $Q \leftarrow Q \cup \{\langle s, i\rangle, \langle s', i - 1\rangle\}$
13  **foreach** $\langle r, i\rangle \in get\_failures()$ **do**
14  **for** $j \in \{0, .., i\}$ **do**
15  $\mathcal{L}_j \leftarrow \mathcal{L}_j \wedge \neg r$
16  $Q \leftarrow \{\langle z, h\rangle \in Q, \texttt{UNSAT}(r \wedge z)$ *or* $\texttt{h} > \texttt{i}\}$
17  **if** $i < k$ **then** $Q \leftarrow Q \cup \{\langle s, i + 1\rangle\}$
18  **for** $i \in \{1, .., k + 1\}$ **do** /* Clause Pushing */
19  CP $\leftarrow \{\langle \neg c, i\rangle \mid c \in \mathcal{L}_{i-1}, c \notin \mathcal{L}_i\}$
20  **while** $CP \neq \{\}$ *or* $workers\_waiting() \neq \{1, .., M\}$ **do**
21  **foreach** $w \in workers\_waiting()$ **do**
22  **if** $CP \neq \{\}$ **then**
23  $\langle \neg c, i\rangle \leftarrow$ pop from $CP$
24  $process\_pushing(w, \mathcal{L}, \langle \neg c, i\rangle)$
25  **foreach** $\langle \neg c, \_\rangle \in get\_failures()$ **do**
26  $\mathcal{L}_i \leftarrow \mathcal{L}_i \wedge c$
27  **if** $\mathcal{L}_{i-1} \equiv \mathcal{L}_i$ **then return** False

2017). Our planning algorithms begin with the partial state $u$ indicated by used assumptions. Then following existing literature on PDR, that candidate reason is further tightened, by further queries to an incremental SAT procedure. Specifically, we iteratively test whether each literal can be removed from $u$ so that the obligation $\langle u, i\rangle$ remains unsatisfied. Iteration associated with reason minimisation proceeds according to a global lexicographic order over literals. A literal is tested for removal by using an incremental SAT solver to determine if the obligation without that literal is still unsatisfiable. Where the obligation is unsatisfiable, the 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. Once all tests are complete, the remaining cube is taken forward in PDR as the reason.

We can now tie these ideas to the pseudocode of Algorithm 1. Taking $M = 1$ we describe serial PDR, and $M > 1$ provides the control flow regarding how the PS-PDR orchestrator coordinates with workers to implement a distributed PDR 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 $k$ 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 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 on $Q$ 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.

The $process\_obligation$ call, 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 records. For PS-PDR, the work of processing an obligation is assigned to, and then completed by an available worker process. 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 $s'$ from the obligation $\langle s, i\rangle$ is found by a worker, as described above, the tuple $\langle s', s, i\rangle$ is sent from the worker to the orchestrator and stored in a buffer. This buffer is accessed by the orchestrator and then emptied, a step occurring at the call to $get\_successes$. A valid successor may not exist, in which case the worker derives a minimised reason $r$ for that, by iteratively calling an incremental SAT procedure as described above. Here, a tuple $\langle r, i\rangle$ is sent to the orchestrator and stored in a separate reasons buffer. The reasons buffer is processed by the orchestrator, which occurs at the call $get\_failures$. The orchestrator understands what workers are available to process obligations via $workers\_waiting$, which returns the set of available worker IDs. Calls to $process\_pushing$ behave 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.

Evaluated obligations that are satisfiable are retrieved and processed on Lines 10-12. Where $i$ is the layer index of an obligation, if $i - 1 = 0$, then $s' \vDash \mathcal{L}_0$, and as $\texttt{SAT}(\mathcal{L}_0 \wedge \texttt{G})$ by definition, $s'$ is a goal state. All states mentioned in queued obligations are either the initial state, or a state that can be reached from the initial state. Thus, in case $i = 1$ on Line 11, the algorithm is able to return *True* because a goal state reachable from $I$ is found. For $i > 1$, the obligation processed by a worker and the successor obligation it computed are both added to the queue $Q$ on Line 12.

Minimal reasons computed by workers are retrieved and processed on Lines 13-17. Any reason calculated by a worker processing an obligation at layer $i$ is conjoined with the layer formulae $j \in [0..i], \mathcal{L}_{i-j}$, maintained by the orchestrator. On Line 16, the orchestrator removes obligations in $Q$ comprising states forbidden by added reasons. Such obligations are guaranteed to be unsatisfiable, therefore can be safely removed. Additionally, *obligation rescheduling* is performed. If $i < k$, then the queried obligation is added back to the queue with a larger layer index, on Line 17. Obligation rescheduling is not necessary for completeness, but is important for performance in practice.

Once the queue is empty and all workers are free, then *clause pushing* is performed and a test for termination is evaluated. 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. Pseudocode for this logic is on Lines 19-26. 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 layers are equivalent. If they are then the problem has no solution, so $False$ is returned on Line 27. Note this equivalence check is a syntactic check, of whether the set of clauses in adjacent layer formulae are identical.

## 4 Parallel Decomposition PDR

Formally, 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 of subproblems is defined and then solved, with these determined efficiently according to a current LADG. The listed position of a subproblem identifies what goals must be achieved, and also the initial conditions that are obligated to be maintained by a valid plan for that subproblem. 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.

We assume we are provided with a binary mutex relation, identifying pairs of facts that cannot be simultaneously true in a problem state. 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 $\Gamma$. 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 $\Gamma$ for later iterations is described below. The subproblems posed at an iteration are defined as follows.

- Being directed and acyclic, $\Gamma$ induces a natural partial order over $\mathbf{P}$. Given $\Gamma$, PD-PDR begins an iteration by creating a list, $\Delta_G$, comprised of all the elements in $\mathbf{P}$ labelled with a goal fact. The only ordering constraint is that a path from $\mathbf{p}_1$ to $\mathbf{p}_2$ in $\Gamma$ constrains $\mathbf{p}_1$ to occur before $\mathbf{p}_2$ in $\Delta_G$, and otherwise the order can be taken as arbitrary.
- For $\mathbf{p}_i \in \Gamma$, let $V(\mathbf{p}_i)$ be the facts labelling $\mathbf{p}_i$.
- Each $\mathbf{p}_G$ in $\Delta_G$ is associated with a set of facts $F(\mathbf{p}_G)$:

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

  $F(\mathbf{p}_G)$ contains most facts that are required to plan for the goals in $\mathbf{p}_G$.
- Missing from $F(\mathbf{p}_G)$ are *exclusions*:

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

  Exclusions are $\mathbf{p}_G$ related facts whose truth value can only change once in a plan.
- Let $\overline{M}(I, \mathbf{p})$ be all true facts from the initial state $I$ that are not in a binary mutex relation with facts in $G \downharpoonright V(\mathbf{p})$. In words, the set of initial state facts that can be true in a state where the goal, projected to facts in $V(\mathbf{p})$, is satisfied. Let $i(\mathbf{p}_G)$ be the integer index of where entry $\mathbf{p}_G$ occurs in list $\Delta_G$. Dependants $\Phi(\mathbf{p}_G)$ are:

$$\begin{aligned} \Phi(\mathbf{p}_G) \quad \equiv \quad & \{f | \mathbf{p}' \in \Delta_G, i(\mathbf{p}') > i(\mathbf{p}_G), f \in F(\mathbf{p}')\} \\ & \cap F(\mathbf{p}_G) \cap \overline{M}(I, \mathbf{p}_G) \end{aligned}$$

  $\Phi(\mathbf{p}_G)$ are facts that are co-dependent on goals mentioned after $\mathbf{p}_G$ in $\Delta_G$. We see in a moment that intersection with $\overline{M}(I, \mathbf{p}_G)$ is to ensure the subproblem associated with a $\mathbf{p}_G$ in $\Delta_G$ is not trivially unsolvable given available mutex information.
- Let $X(\mathbf{p}_G) = F(\mathbf{p}_G) \cup Ex(\mathbf{p}_G)$. Each element $\mathbf{p}_G \in \Delta_G$ then gives a subproblem:

$$\langle X(\mathbf{p}_G), A \downarrow X(\mathbf{p}_G), I \downharpoonright X(\mathbf{p}_G), G \downharpoonright V(\mathbf{p}_G) \wedge I \downharpoonright \Phi(\mathbf{p}_G)\rangle$$

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.

That concatenation operation might 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 fact that can only be changed once. If concatenation fails to yield a valid plan, then a fact not mentioned in the subproblem goal due to an exclusion, or mutex, is identified as *problematic*. PD-PDR then enters a new iteration, creating a corresponding new $\Gamma$ by contracting vertices, and maybe adding facts to the labelling of vertices, from the $\Gamma$ used in

this iteration. The details of the contractions are described below: Each problematic fact already labels an LADG vertex, or otherwise can be added as the sole label of a new LADG vertex with arcs as per the dependency graph (it may not label an LADG vertex yet, because it has been removed via the construction of the PSDG). The LADG vertex the problematic fact labels is contracted with all LADG vertices labelled with facts in a mutex relationship with that problematic fact. 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.

For each problematic fact, should the above operation not result in 2 or more vertices being contracted—e.g., the vertex labelled with the problem fact has no descendants or facts in mutex relationships—then the vertex labelled with the problematic fact 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 $\Gamma$ used in this iteration.

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 $\Phi(\mathbf{p}_G)$ 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 fact in the subproblem are contracted to a single vertex. Additionally, for each unsatisfiable subproblem, should the above operation not result in 2 or more vertices being contracted, then the sole vertex labelled with facts 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 $\Delta_G$.

In review, PD-PDR first takes a concrete planning problem and produces a PSDG, which shows how particular facts depend on other facts. This is used to produce a LADG $\Gamma$ for the initial iteration, which groups facts into components, and shows how the components depend on each other. At each iteration $\Delta_G$ is then constructed, which contains all the vertices in the current LADG which contain facts mentioned in the concrete goal, with an order that satisfies the partial order imposed by the edges of the LADG. Each vertex in $\Delta_G$ represents a section of the goal condition, and 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 not all) of the facts needed to plan for the $\mathbf{p}_G$ portion of the goal. $F(\mathbf{p}_G)$ is the union of all facts mentioned in $\mathbf{p}_G$, with all facts mentioned in descendants of $\mathbf{p}_G$ in $\Gamma$. By taking the descendants, this ensures all facts that are depended upon are included in the subproblem facts, and only the facts which are not depended upon are excluded. Next the exclusions, $Ex(\mathbf{p}_G)$, are found. These are facts which can only change polarity once, often representing a finite resource. Because of this, if more than one subproblem modifies one of these facts, then the concatenation of subproblem plans will be invalid. Excluded facts may be necessary for finding a subproblem plan. To deal with exclusions, PD-PDR may initially allow multiple subproblems to modify an exclusion fact. Then, if multiple subproblem plans modify it, subproblems are "merged" until only one subproblem is able to modify it. We find this technique works well in practice.

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 $\Phi(\mathbf{p}_G)$. $\Phi(\mathbf{p}_G)$ is somewhat similar to $F(\mathbf{p}_G)$, but contains all the facts relevant to subproblems later indicated by $\Delta_G$. For the last subproblem, $\Phi(\mathbf{p}_G) = \emptyset$.

After the series of subproblems is created, each is solved independently in parallel. This results in a few different cases: *(i)* All subproblems have solutions, and no 2 subproblems changed the polarity of the same exclusion – i.e. the concatenation of subplans forms a valid plan for the concrete problem. Then this concatenation is returned as a valid plan for the concrete problem. *(ii)* There is a sole subproblem (corresponding to the concrete problem) which is unsolvable, in which case the concrete problem has no solution. *(iii)* All subproblems have solutions, but multiple subproblems changed the polarity of the same exclusion(s) – i.e. the concatenation of subplans does not form a valid plan for the concrete problem. Then vertices in this $\Gamma$ are merged to create a new $\Gamma$ for the next iteration, and a new series of subproblems is created. This is done so that separate subproblems in this new series have less of an ability to modify the same exclusion. *(iv)* One or more subproblems does not have a solution. Then similarly to case *(iii)*, vertices in this $\Gamma$ are merged to create a new $\Gamma$ and a new series of subproblems is created. This may loosen the subproblems goal conditions, making it feasible for subproblems to be solved, without compromising the correctness. Both these last two cases result in strictly fewer vertices in the new $\Gamma$. There is then a finite number of iterations, as the first $\Gamma$ has finite vertices, each successive $\Gamma$ must have at least one vertex, and the number of vertices in $\Gamma$ in each iteration decreases. Because of this, and that each iteration takes a finite amount of time, we can guarantee that the algorithm will terminate.

**Example 2** (Performing PD-PDR on a Logistics Problem)**.** *Consider a Logistics problem $\langle X, A, I, G\rangle$ using $X$ and $A$ from Example 1, and where:*
$I^+ = \{at(P1, L1), at(P2, L1), at(T, L1)\}$*:*

$$\begin{aligned} G &= at(P1, L1) \wedge at(T, L2) \\ I &= \bigwedge_{x \in I^+} x \wedge \bigwedge_{x \in X, x \notin I^+} \neg x \end{aligned}$$

*To perform* PD-PDR*, we first compute* $\Gamma = (\{\mathbf{p}_1, \mathbf{p}_2\}, \{(\mathbf{p}_1, \mathbf{p}_2)\})$ *where:*

$$\begin{aligned} V(\mathbf{p}_1) &= \{at(P1, L1), at(P1, L2), in(P1, T)\} \\ V(\mathbf{p}_2) &= \{at(T, L1), at(T, L2)\} \end{aligned}$$

*No propositions which mention $P2$ are included, as while they are in the dependency graph, they are not in the PSDG. We then compute: $\Delta_G = [\mathbf{p}_1, \mathbf{p}_2]$. All SCCs are mentioned as they all include propositions mentioned in $G$. $\mathbf{p}_1$ occurs before $\mathbf{p}_2$ as there is a path $\mathbf{p}_1 \overset{\Gamma}{\rightsquigarrow} \mathbf{p}_2$.*

$$\begin{array}{lcl} F(\mathbf{p}_1) & = & \{at(P1,L1), at(P1,L2), in(P1,T), \\ & & at(T,L2), at(T,L2)\} \\ F(\mathbf{p}_2) & = & \{at(T,L1), at(T,L2)\} \\ \Phi(\mathbf{p}_1) & = & \{at(T,L1), at(T,L2)\} \\ Ex(\mathbf{p}_1) & = & Ex(\mathbf{p}_2) = \Phi(\mathbf{p}_2) = \{\} \end{array}$$

*The two subproblems are then:*

- $\langle X_1, A_1, I_1, G_1 \rangle$ *where:*

$$\begin{array}{lcl} X_1 & = & \{at(P1,L1), at(P1,L2), in(P1,T), \\ & & at(T,L1), at(T,L2)\} \\ A_1 & : & \textit{Can be described by the actions:} \\ & & \textit{Load } P1 \textit{ into } T \textit{ from } L1 \textit{ or } L2, \\ & & \textit{Unload } P1 \textit{ from } T \textit{ to } L1 \textit{ or } L2, \\ & & \textit{Drive } T \textit{ between } L1 \textit{ and } L2 \\ I_1 & = & at(P1,L1) \wedge \neg at(P1,L2) \wedge \neg in(P1,T) \wedge \\ & & at(T,L1) \wedge \neg at(T,L2) \\ G_1 & = & at(P1,L2) \wedge at(T,L1) \wedge \neg at(T,L2) \end{array}$$

- $\langle X_2, A_2, I_2, G_2 \rangle$ *where:*

$$\begin{array}{lcl} X_2 & = & \{at(T,L1), at(T,L2)\} \\ A_2 & : & \textit{Can be described by the actions:} \\ & & \textit{Drive } T \textit{ between } L1 \textit{ and } L2 \\ I_2 & = & at(T,L1) \wedge \neg at(T,L2) \\ G_2 & = & at(T,L2) \end{array}$$

*Two corresponding example subproblem plans are:*

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

## 5 Experiments

We implemented PS-PDR and PD-PDR planners in C++, using the MADAGASCAR (Rintanen 2012) codebase for parsing and preprocessing. We also implemented a serial PDR planner, called PDR-S, and a distributed PDR portfolio, called PDR-P.[3] LINGELING (Biere 2010) is used as the incremental SAT solver in all our systems. PDR-P operates by having n distinct serial PDR processes running in parallel. These processes share some information: *(i)* Clauses found by one process are given to all other processes - strengthening each other's layer information, and *(ii)* Their $k$-values (PDR upper layer number) are synchronized. When progressing to the next layer (incrementing $k$), instead of pushing layer clauses independently, this work is distributed amongst the workers as their layer information is synchronized.

Our motivations for implementing the comparison algorithms, PDR-S and PDR-P planners, are threefold: *(i)* we isolate the reasons for performance gaps, employing the same data structures and SAT decision procedure across all implementations, *(ii)* 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, and *(iii)* 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 "unplannability" benchmarks from the 2016 IPC unplannability track, which are *unsatisfiable* – i.e., they are designed with the intent that problems will not have solutions. We exclude reporting on some unplannability domains, namely: BAG-BARMAN, BAG-GRIPPER, OVER-TPP and SLIDING-TILES, because no systems evaluated are able to produce UNSAT proofs 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 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. PD-PDR candidate concrete plan validity is determined using VAL (Howey, Long, and Fox 2004). We evaluate PD-PDR, along with all other PDR systems, on all

[3] Source available at https://doi.org/10.5281/zenodo.8020680. Updated version may be available at: https://github.com/ANU-HPC/parallel-pdr

satisfiable planning benchmarks that we know to be susceptible to compositional analysis.

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. We exclude reporting on PD-PDR runtimes on domains which do not meet this criteria, as when there is only one subproblem, PD-PDR behaves as PDR-S with some additional overheads. 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 Figure 3. Note, the number of solved instances in the PD-PDR plot is relatively low, because we only report instances amenable to decomposition. Figure 2 features the cumulative number of problems solved, as a function of time, for each solver. There are separate figures, with and without PS-PDR. The figure excluding PS-PDR considers all problems, and the figure including PD-PDR only includes problems where PD-PDR is applicable.

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 ∼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.

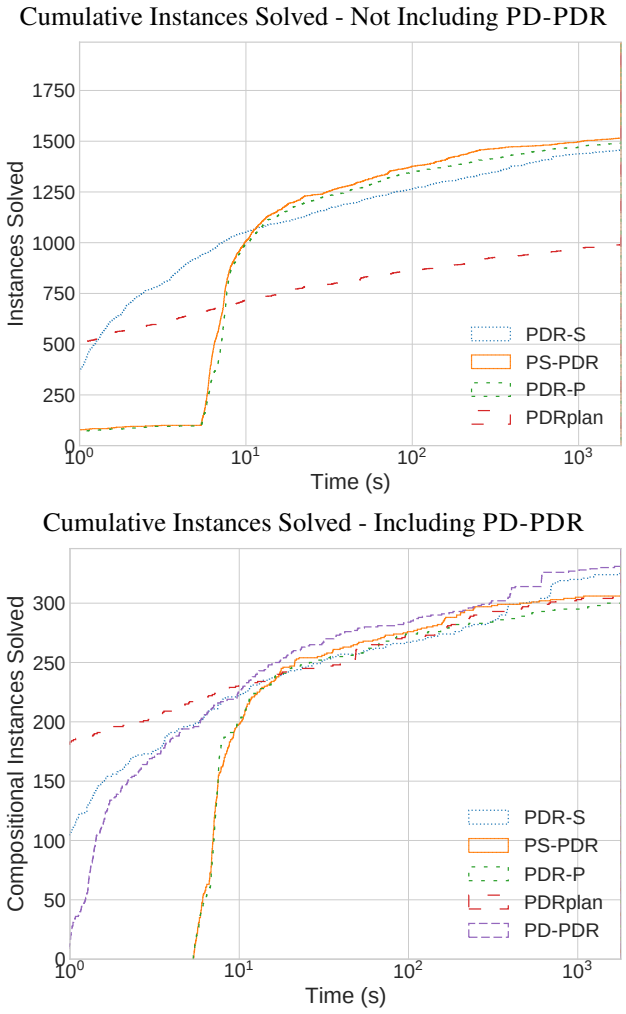


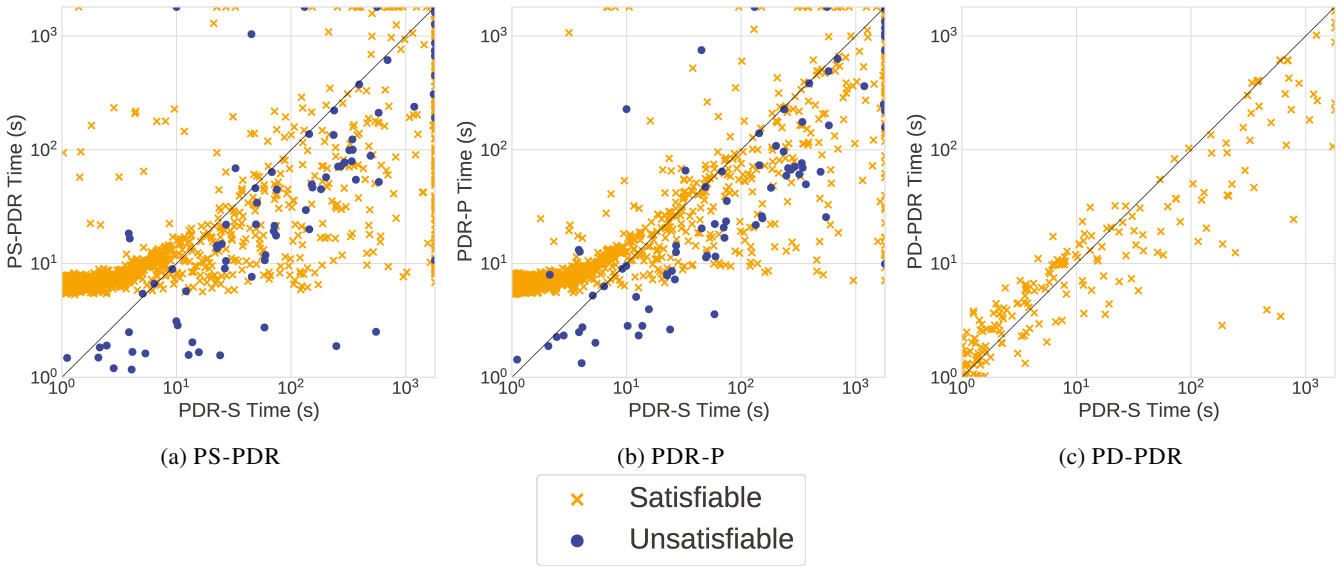

Figure 2: Cumulative Number of Instances Solved

## 6 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 reasoning about problem *safety* without explicitly unrolling the transition relation, Property Directed Reachability was a phrase later coined in (Eén, Mishchenko, and Brayton ). 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. 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. 1, Lines 18-27) can be performed by one serial process, and problem *safety* (Alg. 1, Line 27) can be detected by that process where applicable. In this original work, search diversity is enhanced using a nondeterministic SAT-solver ZCHAFF (Vizel, Weissenbacher, and Malik 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. With these modifications, Bradley was able to demonstrate that a portfolio using 12 cores can complete an additional 12 proofs in HWMCC'10 competition

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




(a) PS-PDR (b) PDR-P (c) PD-PDR

Figure 3: Scatterplots Comparing Problem Runtimes Between Each New Solver and the Baseline PDR-S.

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, Gurfinkel, and Chaki 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.

Our approach to decompositional planning using PDR is 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 causal 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. 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, Norrish, and Gretton 2015), a work in which the authors employ a goal editing idea to plan in systems composed of a set of symmetric subsystems.

## 7 Conclusions

We are the first to develop and evaluate distributed PDR systems for planning. Our PS-PDR algorithm improves on existing PDR portfolio algorithms by: *(i)* using obligation rescheduling, and *(ii)* using a centralised queue. Distinct from existing decompositional approaches, our PD-PDR algorithm analyses the structure of problems to break them into subproblems which are solved independently. We find our additions, PD-PDR and PS-PDR are generally able to outperform baseline PDR. Additionally, using PS-PDR in place of PDR-S in PD-PDR may wield even greater performance increases than we have measured so far.

## Acknowledgements

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.

## References

Abdulaziz, M., and Berger, D. 2021. Computing plan-length bounds using lengths of longest paths. In *Thirty-Fifth AAAI Conference on Artificial Intelligence 2021*, 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, IJCAI*, 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):485–520.

Amir, E., and Engelhardt, B. 2003. Factored planning. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, 929–935. Morgan Kaufmann.

Baumgartner, J.; Kuehlmann, A.; and Abraham, J. A. 2002. Property checking via structural analysis. In *Computer Aided Verification, 14th International Conference, CAV*, volume 2404 of *Lecture Notes in Computer Science*, 151–165. Springer.

Biere, A.; Cimatti, A.; Clarke, E. M.; and Zhu, Y. 1999. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS*, volume 1579 of *Lecture Notes in Computer Science*, 193–207. Springer.

Biere, A. 2010. Lingeling, plingeling, picosat and precosat at sat race 2010. *FMV Technical Report 10/1*. Institute for Formal Models and Verification, Johannes Kepler University.

Bradley, A. R. 2011. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference*, volume 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.

Burgess, M. A.; Gretton, C.; Milthorpe, J.; Croak, L.; Willingham, T.; and Tiu, A. 2022. Dagster: Parallel structured search with case studies. In *PRICAI 2022: Trends in Artificial Intelligence - 19th Pacific Rim International Conference on Artificial Intelligence*, volume 13629 of *Lecture Notes in Computer Science*, 75–89. Springer.

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

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

Eén, N., and Sörensson, N. 2003. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, 502–518. Springer.

Eén, N.; Mishchenko, A.; and Brayton, R. K. Efficient implementation of property directed reachability. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD*, 125–134. FMCAD Inc.

Eriksson, S., and Helmert, M. Certified unsolvability for SAT planning with property directed reachability. In Beck, J. C.; Buffet, O.; Hoffmann, J.; Karpas, E.; and Sohrabi, S., eds., *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling*.

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):189–208.

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

Gocht, S., and Balyo, T. 2017. Accelerating SAT based planning with incremental SAT solving. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS*, 135–139. AAAI Press.

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

Huang, J., and Darwiche, A. 2003. A structure-based variable ordering heuristic for SAT. In Gottlob, G., and Walsh, T., eds., *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 1167–1172. Morgan Kaufmann.

Kautz, H. A., and Selman, B. Pushing the envelope: Planning, propositional logic and stochastic search. In Clancey, W. J., and Weld, D. S., eds., *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference*, 1194–1201.

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

Kelareva, E.; Buffet, O.; Huang, J.; and Thiébaux, S. 2007. Factored planning using decomposition trees. In Veloso, M. M., ed., *IJCAI 2007, 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):243–302.

Komuravelli, A.; Gurfinkel, A.; and Chaki, S. 2016. Smt-based model checking for recursive programs. volume 48, 175–205.

Kroening, D.; Ouaknine, J.; Strichman, O.; Wahl, T.; and Worrell, J. 2011. Linear completeness thresholds for bounded model checking. In Gopalakrishnan, G., and Qadeer, S., eds., *Computer Aided Verification - 23rd International Conference, CAV*, volume 6806 of *Lecture Notes in Computer Science*, 557–572. Springer.

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

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.

Rintanen, J., and Gretton, C. O. 2013. Computing upper bounds on lengths of transition sequences. 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):1031–1080.

Rintanen, J. 2004. Evaluation strategies for planning as satisfiability. In de Mántaras, R. L., and Saitta, L., eds., *Proceedings of the 16th Eureopean Conference on Artificial Intelligence*, 682–687. IOS Press.

Rintanen, J. 2008. Regression for classical and nondeterministic planning. In Ghallab, M.; Spyropoulos, C. D.; Fakotakis, N.; and Avouris, N. M., eds., *ECAI 18th European Conference on Artificial Intelligence*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, 568–572. IOS Press.

Rintanen, J. 2012. Planning as satisfiability: Heuristics. *Artificial intelligence* 193:45–86. Publisher: Elsevier.

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

Streeter, M. J., and Smith, S. F. 2007. Using decision procedures efficiently for optimization. In Boddy, M. S.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, 312–319. AAAI.

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

Vizel, Y.; Weissenbacher, G.; and Malik, S. 2015. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE* 103(11):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, IJCAI*, 1178–1185. Morgan Kaufmann.