Counter-Example Based Planning Under Uncertainty

Xiaodi Zhang

A thesis submitted for the degree of

Doctor of Philosophy

The Australian National University

School of Computing

April 2025

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

Disclaimer

This dissertation is an account of research undertaken between August 2021 and May 2025 at the Research School of Computing, The Australian National University, Canberra, Australia.

Except where acknowledged in the customary manner, all material presented in this dissertation including figures are original and have not been submitted in whole or part for a degree in any university.

This thesis is based in part on papers published in refereed journal or conferences, or under review at such a conference or journal. The development of ideas and research was undertaken with guidance from my supervisors Charles Gretton and Alban Grastien, and published papers that describe research in this thesis were written collaboratively with them. Below I describe my contributions to the research in each chapter and associated papers.

1. Chapter 3 includes work from two papers:

   - Zhang, X., Grastien, A., & Scala, E. (2020, April). Computing superior counter-examples for conformant planning. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 34, No. 06, pp. 10017-10024).
   - Zhang, X., & Grastien, A. (2023, July). Improvements to CPCES. In *Proceedings of the International Symposium on Combinatorial Search* (Vol. 16, No. 1, pp. 110-118).

   My research forms a major contribution to all sections of two papers.

2. Chapter 4 includes work from the paper:

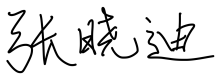
   - Zhang, X., Grastien, A., & Gretton, C. (2024, May). A counter-example based approach to probabilistic conformant planning. In *Proceedings of the International Conference on Automated Planning and Scheduling* (Vol. 34, pp. 689-697).

   My research forms a major contribution to all sections of the paper.

3. Chapter 5 includes work from the paper, which is currently under review:

- Zhang, X., Grastien, A., & Gretton, C. Counter-example guided conformant and probabilistic conformant planning. *Journal of Artificial Intelligence Research*.

My research forms a major contribution to all sections of the paper.



Xiaodi Zhang

13 October 2025

Acknowledgements

During my PhD studies, I have received generous help and support from many people.

First and foremost, I would like to express my deepest gratitude to my supervisors, Dr. Alban Grastien and Dr. Charles Gretton. Their guidance has been instrumental in shaping the direction of my research and supporting me throughout every stage of my PhD journey. Beyond their academic supervision, they have also provided invaluable encouragement and thoughtful care in my daily life. Without their support, the completion of this thesis would not have been possible.

I would also like to thank the other members of my supervisory panel, Dr. Pascal Bercher and Dr. Hanna Kurniawati, for their insightful feedback and regular evaluations of my research progress and quality over the years. I am particularly grateful to Dr. Pascal Bercher for his helpful suggestions on revising this thesis.

My sincere thanks also go to Dr. Jinbo Huang for his guidance on the work presented in Section 4.4 of this thesis.

Most importantly, I would like to thank my parents for their unwavering support—both emotionally and practically—throughout this journey. Their love and encouragement have been the foundation of my perseverance.

Abstract

Planning under uncertainty broadly involves finding a plan for an agent acting in an uncertain environment. Conformant Planning (CP) and Probabilistic Conformant Planning (PCP) are two specific and related planning problems that involve uncertainty. In CP, an agent that is unable to observe the environment must find a plan that achieves its goal under a qualitatively uncertain initial state. CP can involve uncertainty either in the initial state alone or in both the initial state and the action effects. PCP features quantitative uncertainty, extending CP by introducing a probability distribution over the initial state alone or over both the initial state and the action effects, and requiring the goal to be reached with at least a given probability threshold. In this thesis, the CP and PCP problems involve uncertainties in their initial states only, while action effects are assumed to be deterministic.

A key milestone in the development of CP algorithms was by Grastien and Scala (2020). Those authors proposed `CPCES`, a counter-example based approach to CP with uncertain initial states only. `CPCES` starts with an empty candidate plan, and then iteratively: (1) finds a counter-example — an initial state where the candidate plan fails — and (2) re-plans to satisfy all previously discovered counter-examples. This process continues until a valid plan is found or the problem is proven unsolvable. The basis of planning in `CPCES` is to abstract a CP problem into a corresponding classical planning problem, which serves as a simplified deterministic model for guiding the search.

This thesis first explores three methods to improve the runtime performance of the `CPCES` algorithm. The first identifies *certain facts*, whose values remain known with certainty during the plan execution. Certain facts are represented explicitly in the classical abstraction. Our innovation, the merging of certain facts in the classical abstraction, substantially reduces the size of that abstraction thereby lowering the cost of reasoning about it. The second, warm-starting `CPCES`, identifies *important initial states* before commencement of iterative refinement, enabling `CPCES` to sometimes find a valid plan immediately, or converge in fewer iterations. The third integrates the `Fast Downward System` (Helmert 2006) into `CPCES`, developing a bespoke stratified compilation approach for synthesizing successive classical

abstractions. Experimental results show that all three methods significantly improve baseline CPCES efficiency.

Moving our attention to quantitative uncertainty, we reimagine CPCES to develop a counter-example driven algorithm to solve PCP. Like CPCES, our approach iteratively searches for candidate plans via a classical abstraction, and then counter-examples for such plans. Instead of using a single initial state as a counter-example, we introduce *counter-tags*, a concept that enables the algorithm to reason over succinct representations of sets of initial states. Our algorithm searches over counter-tags, computing their probabilities efficiently by using Deterministic Decomposable Negation Normal Form (d-DNNF) representation of discrete probability distributions. We call our algorithm p-CPCES. To address the challenge of disjunctive goals posed by classical abstractions encountered using p-CPCES, we further develop a hitting set approach to eliminate disjunctive goals from classical abstractions, thereby removing the need for classical planners capable of handling such goals. This enhanced version of the algorithm is called p-CPCES-hit. Experiments show p-CPCES-hit outperforms the state-of-the-art PCP planner Probabilistic-FF (Domshlak and Hoffmann 2006, 2007). The good performance of p-CPCES-hit is particularly pronounced on complex problems, where it is faster than the incumbent algorithms.

Work performed by p-CPCES is amenable to acceleration by exploiting parallel computation. We introduce parallel-CPCES, a multi-core approach to solving PCP. The p-CPCES-hit algorithm involves three sequential steps per iteration: (1) computing counter-tags, (2) computing their probabilities, and (3) solving for a candidate plan using a classical abstraction. Any step can become a bottleneck if it takes too long and parallel execution mitigates this issue. We implement a framework combining p-CPCES-hit, a database, and a file system, where results from any stage are stored in a database, and subprocesses communicate efficiently via the file system. Each stage is executed on multiple CPUs to improve system walltime efficiency. Additionally, we introduce a new hitting set strategy highly suited to our parallel setting to further enhance planning performance. Experimental results confirm that parallel-CPCES finds solutions faster than p-CPCES-hit, with speed improving as more workers (i.e., CPUs) are added.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Planning lies at the heart of artificial intelligence, enabling agents to generate sequences of actions that achieve goals (Russell and Norvig 2020). In many real-world domains—such as autonomous driving, medical diagnosis, disaster response, and human–robot collaboration—agents must act under incomplete information and stochastic outcomes. While conformant planning, probabilistic conformant planning, and contingent planning have been proposed to address these challenges, counter-example based approaches have so far been applied primarily to conformant planning (Kurien et al. 2002; Nguyen et al. 2012), and their effectiveness in other planning paradigms remains largely unexplored. Counter-example based planning offers a promising alternative: instead of reasoning over the entire belief space, it incrementally constructs plans by focusing only on sampled counter-examples that refute candidate plans (or policies). This not only mitigates the scalability bottleneck but also opens the possibility of integrating machine learning techniques to guide sampling and improve efficiency. Exploring this direction is crucial, as it may yield more scalable and robust planning techniques with broad impact across domains where uncertainty is inherent.

In this chapter, we first introduce three important planning paradigms: classical planning, conformant planning, and probabilistic conformant planning. We then present `CPCES`, a counter-example based conformant planner, and introduce the hitting set strategy. Finally, we provide an overview of the thesis structure and contributions.

1.1 Backgrounds

In this section, we provide the background knowledge required for understanding this thesis. We begin by presenting the formal definitions of classical planning, conformant planning, and probabilistic conformant planning. We then introduce `CPCES`, a counter-example based conformant planner that forms the foundation of the subsequent research. We finally introduce hitting set strategy that will be used in our counter-example based PCP planner.

1.1.1 Classical Planning

Classical planning is a fundamental area in artificial intelligence. It involves searching for a plan (a sequence of actions) in a fully known environment, where executing the plan enables an agent to transition from the initial state to the goal state. In classical planning, there is a unique initial state and some goal states, and the effects of each action are deterministic. Classical planning has been widely applied across various industrial domains. For example, in transportation and supply chain management, it facilitates the planning of efficient logistics and supply routes. In robotics, classical planning is used to control robots, enabling them to complete tasks.

Classical planning is a computationally complex problem. When the plan length is not bounded, solving a classical planning problem is PSPACE-COMPLETE. This is mainly because classical planning involves searching for an action sequence in a discrete state space, which can be exponentially large: if there are n facts, the size of the state space is 2^n.

1.1.2 Definition of Classical Planning

Definition 1. *A classical planning problem can be represented by a tuple* $P = \langle F, A, I, G \rangle$*, where:*

- F *is a set of Boolean variables (also called propositions), each of which can take values* ***True*** *or* ***False****. A state* S *is an assignment of truth values to all variables in* F*. Equivalently, a state can be represented as the set of variables*

that are assigned ***True*** *under* S*.*

- A *is a set of actions. Each action* $o \in A$ *is a pair* $o = \langle pre(o), \mathit{eff}(o) \rangle$*, where:*
  - *The precondition* $pre(o) \subseteq F$ *specifies the set of variables that must be* ***True*** *for the action* o *to be applicable;*
  - *There are two kinds of effects. The first is deterministic effects* $\mathit{eff}(o)$*, which are typically divided into two disjoint subsets: the positive effects* $\mathit{eff}^+(o)$ *and the negative effects* $\mathit{eff}^-(o)$*. Applying* o *to a state adds all variables in* $\mathit{eff}^+(o)$ *(setting them to* ***True****) and removes all variables in* $\mathit{eff}^-(o)$ *(setting them to* ***False****). The second is conditional effects* $\mathit{coneff}(a) = \langle c, \mathit{eff}^+, \mathit{eff}^- \rangle$*, representing the condition and its corresponding positive and negative effects. An action can have both deterministic effects and conditional effects.*
- $I \subseteq F$ *is the initial state, representing the set of variables that are* ***True*** *initially;*
- $G \subseteq F$ *is the goal condition, which specifies the set of variables that must be assigned* ***True*** *in a state for that state to satisfy the goal.*

A plan for a classical planning problem is a sequence of actions $\pi = a_1 \dots a_n$ such that, starting from the initial state I and applying the actions in sequence, the resulting state satisfies the goal condition.

Example 1. *We use a* `DISPOSE` *problem in Figure 1.1 to illustrate classical planning. In a* 2×2 *grid, a robot is initially positioned at (1,1). A can is located at (2,1), and a trash bin is placed at (2,2). The robot can perform three actions: move, pickup, and dispose. The goal state is to ensure that the can is disposed of in the trash bin. A valid solution to this problem is: move(right), pickup(can), move(up), dispose(can). In this example, the initial state is unique and the environment is fully deterministic.*

1.1.3 Conformant Planning

Conformant Planning (CP) involves finding a sequence of actions that can transition an agent from an initial state to a goal state, without the benefit of observations during plan execution. Unlike classical planning problems, where all

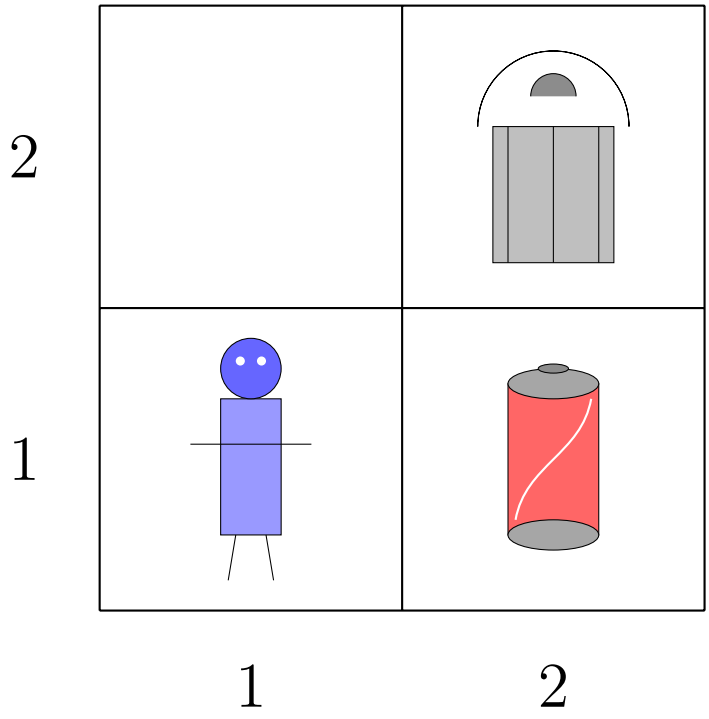


Figure 1.1: A robot initially stands at position (1,1) in a 2 × 2 grid. There is a can located at (2,1) that needs to be picked up and disposed of in a trash bin at (2,2). The robot can perform three actions: move, pickup, and dispose. A valid solution to this problem is: move(right), pickup(can), move(up), dispose(can).

information is deterministic and known, CP models uncertainties. CP can involve uncertainty either in the initial state alone or in both the initial state and the action effects. In this thesis, we focus specifically on CP problems where all uncertainty is in regards to the initial state.

Classical planning typically assumes full certainty about the initial state, which limits its applicability in real-world scenarios where uncertainty is prevalent. In contrast, CP solutions are robust to such uncertainty, ensuring that the goal is achieved regardless of unknown initial conditions. This is especially important in situations where sensors are either expensive, unreliable, or unavailable, making it impractical to continuously gather new information. CP addresses this challenge by assuming the worst-case scenario and devising plans that account for all possible contingencies. For instance, in high-stakes exploration tasks such as space

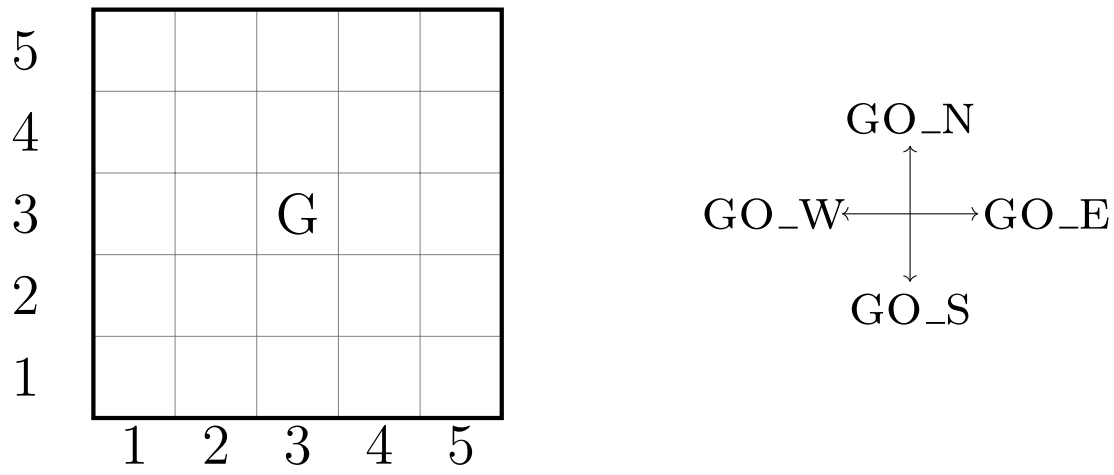


Figure 1.2: Graphical representation of a conformant planning problem. A robot is standing in a 5×5 grid surrounded by walls. Its initial location is unknown. The goal is to move the robot to "G". One valid plan is GO_E $\times$ 4, GO_S $\times$ 4, GO_N $\times$ 2, GO_W $\times$ 2.

missions or underwater expeditions, where obtaining real-time feedback is costly or impossible, CP enables agents to act confidently, knowing that their plans will succeed despite the lack of detailed knowledge about the environment.

Example 2. *To illustrate CP, we use a* `Grid World` *as an example (Figure 1.2), and this thesis explains our method for CP through this scenario. A robot, which lacks observations, is standing in a 5 × 5 grid, but its exact location is unknown. The grid is bordered by walls that prevent the robot from moving outside the grid. The robot has four actions:* GO_S, GO_N, GO_E, *and* GO_W, *representing movements to the South, North, East, and West, respectively. If the robot is blocked by a wall when acting to move in a direction, it will remain stationary. The objective is to move the robot to the center of the grid (3,3), labeled as "G" in Figure 1.2, no matter where the robot is initially. One valid solution for this example is* GO_E *× 4,* GO_S *× 4,* GO_N *× 2, and* GO_W *× 2.*

1.1.4 Definition of Conformant Planning

In deterministic conformant planning, a state is defined as a subset of facts $s \subseteq F$, where each fact in s is **True**. The set F of facts is used to construct the set $\mathcal{L}(F)$, which comprises propositional formulas over F.

Definition 2. *A deterministic conformant planning problem $P = \langle F, N, A, I, G \rangle$ is defined as follows:*

- *F: This represents a finite set of facts.*
- *N: This denotes a set of action names, also known as actions.*
- *A: This function maps each action name in N to a triplet that consists of a precondition $pre(a)$, a set of conditional effects $coneff(a)$. Each element in $coneff(a)$ is a tuple $coneff(a) = \langle c, eff^+, eff^- \rangle$, with c containing only positive facts that must hold true for the effect to apply, eff^+ the set of facts to be added, and eff^- the set of facts to be removed.*
- *I: This is the initial condition, a propositional formula that defines the initial state.*
- *G: This is the goal condition, also a propositional formula representing the goal state.*

A state s is initial if it satisfies the initial condition, written as $s \models I$, where *models* denotes logical entailment — i.e., all facts specified by I hold **True** in s. Similarly, it is deemed a goal state if it satisfies $s \models G$. The application of an action a in state s leads to positive effects $eff^+(s, a)$ which are determined by aggregating the positive effects for all conditions c in $coneff(a)$ that are satisfied in s.

$$eff^+(s, a) = \bigcup_{\substack{\langle c, eff^+, eff^- \rangle \in coneff(a) \\ s \models c}} eff^+.$$

Negative effects are calculated similarly.

$$\mathit{eff}^-(s,a) = \bigcup_{\substack{\langle c, \mathit{eff}^+, \mathit{eff}^- \rangle \in \mathit{coneff}(a) \\ s \models c}} \mathit{eff}^-.$$

The resulting state from applying action a in state s is represented as $s[a]$, where

$$s[a] = s \setminus \mathit{eff}^-(s,a) \cup \mathit{eff}^+(s,a).$$

This operation is valid under two conditions:

1. state s must meet the precondition of the action:

$$s \models \mathit{pre}(a);$$

2. There must be no conflict between the positive and negative effects:

$$\mathit{eff}^+(s,a) \cap \mathit{eff}^-(s,a) = \emptyset.$$

A plan consists of a series of actions $\pi = a_1, \ldots, a_k$. Executing this sequence starting in initial state s_0 results in a progression through states $s_1, \ldots, s_k$, where each state s_i is derived from the previous state s_{i-1} by applying the action a_i : $s_i = s_{i-1}[a_i]$. We denote the result state reached by this sequence as $s_0[\pi]$. A plan is termed applicable in s_0 if each action a_i can be successfully applied in the corresponding state s_{i-1}, and the final state s_k meets the goal condition. A solution, or a conformant plan, is considered valid for a deterministic conformant planning problem if it holds across all initial states. The set of all valid conformant plans for problem P is denoted by $\Pi(P) \subseteq N^*$, where N^* denotes the Kleene closure of the action set N, i.e., the set of all finite sequences (including the empty sequence) composed of actions from N. A conformant planning problem is a *classical* planning problem if there is only one initial state.

Example 3. *Let's refer to the example from Section 1.1.3. A state is represented as a set of facts* $\{x_i, y_j\}$ *where* $i \in \{1, \ldots, 5\}$ *and* $j \in \{1, \ldots, 5\}$. *Here,* x_i *corre-*

sponds to the column number labeled in the grid, and y_j represents the row number. There are a total of 25 possible initial states, and the goal state is $\{x_3, y_3\}$. There are four actions, each without a precondition but with several conditional effects.

1.1.5 Probabilistic Conformant Planning

Compared with Conformant Planning, the uncertainty in Probabilistic Conformant Planning (PCP) involves probability distributions. PCP is a planning problem where the objective is to find a sequence of actions that transition an agent from an initial state to a goal state, under conditions where either only the initial state is uncertain and follows a probability distribution, or both the initial state and the effects of actions are uncertain and governed by probabilities. The agent lacks observational capabilities, and the plan must achieve the goal with a certain probability guarantee. In this thesis, we focus specifically on PCP problems where the initial state is uncertain. Under this assumption, CP can be regarded as a special case of PCP, where the probability distribution for the initial state or action effects is uniform, and the probability threshold for reaching the goal is 1. The need to account for probability distributions during plan search makes PCP significantly more challenging to be solved than CP.

PCP is both useful and applicable in real-world scenarios. First, by incorporating probabilities into the planning process, PCP can optimize for the most likely successful outcomes. This approach allows planners to account for the likelihood of various scenarios and to select actions that maximize the probability of achieving the goal. Second, in complex systems where actions have stochastic effects—such as robotics, autonomous vehicles, or dynamic environments—PCP enables the development of more sophisticated and effective plans. It helps these systems navigate uncertainty and variability in their operational contexts.

Example 4. *We continue using the* `Grid World` *example to illustrate PCP, but with additional constraints. In this scenario (Figure 1.3), a robot without observation stands in a 3×3 grid and aims to move to the center of the grid (2,2) with a probability guarantee of 0.75. The initial location of the robot is uncertain: the probabilities of being at x_1, x_2, and x_3 are 0.2, 0.7, and 0.1, respectively, and the probabilities of being at y_1, y_2, and y_3 are 0.2, 0.7, and 0.1, respectively. The grid*

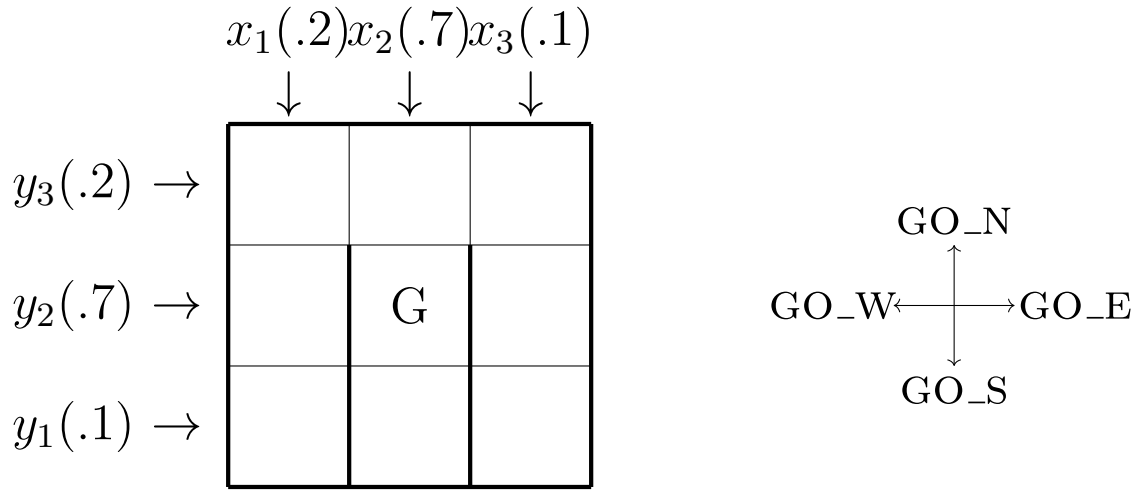


Figure 1.3: Graphical representation of a probabilistic conformant planning problem. The bold lines represent walls that the robot cannot pass through. The robot can only GO_E and GO_W when it is in the first row (y_3) of the grid, as there are vertical walls on y_1 and y_2 that block horizontal movement. The robot's initial location is uncertain, with its initial vertical and horizontal positions indicated by the probabilities labeling the rows and columns, respectively (e.g., the probability of starting at (x_1, y_2) is $0.2 \times 0.7 = 0.14$). The objective is for the robot to reach the center location with a probability of at least 0.75.

is bordered by walls (bold lines) that prevent the robot from moving outside, and if the robot encounters a wall, it will remain stationary. The robot can only move East or West when it is in the top row (y_3) of the grid. There are vertical walls on y_1 and y_2 that block horizontal movement. Notice that there are several differences compared to Figure 1.2. First, in this example, the initial state is represented by a probability distribution. Second, the grid size is 3×3, not 5×5 as in the CP example. Third, in this example, the robot is only allowed to move horizontally when positioned at y_3 (the precondition of GO_E *and* GO_W *is $\neg y_3$). This setting is designed to demonstrate how to handle action preconditions during problem projection.*

A valid solution to this problem is GO_N, GO_W, GO_E, GO_S. *This plan covers four positions: (1,3), (1,2), (2,3), (2,2), resulting in a probability of reaching the goal state of 0.04 + 0.14 + 0.14 + 0.49 = 0.81, which is greater than or equal to the required 0.75.*

1.1.6 Definition of Probabilistic Conformant Planning

To better describe PCP algorithms in this thesis, we will use propositional formulas (this is different from the way CP is defined) to represent some elements in a PCP problem. Given a propositional formula φ, we write $vars(\varphi)$ to denote the set of propositional variables in φ. Below, formulas are understood as a conjunction $\phi = \bigwedge \phi_i$; they may consist of a single conjunct.

Definition 3. *A probabilistic conformant planning problem $\mathcal{P} = \langle F, A, I, G, \tau \rangle$ is defined as follows:*

- *F: A finite set of facts, where a state is represented as a subset $s \subseteq F$, with each fact in s being **True**. This state can be understood as a propositional formula $\bigwedge_{f \in s} f \wedge \bigwedge_{f \in F \setminus s} \neg f$.*
- *A: A set of actions, where each action $a \in A$ is a tuple $\langle name(a), pre(a), coneff(a) \rangle$, consisting of a precondition $pre(a)$ and a set of conditional effects $coneff(a) = \{\langle con_1, eff_1^+, eff_1^- \rangle, \ldots, \langle con_k, eff_k^+, eff_k^- \rangle\}$. The precondition is a conjunction of facts over F and is **True** if trivial. In each conditional effect $\langle con, eff^+, eff^- \rangle$, con, called the condition, and two sets of facts, eff^+ and eff^-, representing the positive and negative effects, respectively. Action a is applicable in state s if $s \models pre(a)$. The positive effects of a in s are $Pos(a, s) = \bigcup_{\langle con, eff^+, eff^- \rangle \in coneff(a), s \models con} eff^+$, and the negative effects $Neg(a, s)$ are defined similarly. The resulting state s' after executing action a is $s'[a] = s \cup Pos(a, s) \setminus Neg(a, s)$.*
- *I: The initial probability function $I : 2^F \to [0, 1]$. If the only initial state is i, it is represented as $I = \{i\}$ rather than $I = \{i \mapsto 1\} \cup \{s \mapsto 0 | s \neq i\}$. The function I is extended to sets of states as $I(S) = \sum_{s \in S} I(s)$.*
- *G: The goal condition, expressed as a conjunction of facts. It represents the goal the PCP algorithm aims to achieve.*
- *τ: The* probability threshold*, a rational number between 0 and 1.*

A plan $\pi = name(a_1) \ldots name(a_k)$ is a sequence of action names. Executing this sequence starting from an initial state s_0 results in a progression through states $s_1, \ldots, s_k$, where each state s_i is obtained by applying the action a_i to the previous state s_{i-1}, i.e., $s_i = s_{i-1}[a_i]$.We generalize this notation by writing $s[\pi]$ to denote

the state resulting from executing the plan π starting from state s. A plan π is *valid* for an initial state s_0 if each action a_i can be successfully applied in the corresponding state s_{i-1}, and the final state s_k satisfies the goal condition, i.e., $s_k = s_0[\pi] = G$. We denote the set of states in which π is valid as $[\![\pi]\!]$. A plan is valid for the problem $\mathcal{P}$, written $\pi \in \Pi(\mathcal{P})$, if it holds for some initial states and the probability of these initial states is no less than τ: $I([\![\pi]\!]) \geq \tau$.

Example 5. *Let's refer to the example from Section 1.1.5. This example includes six facts:* $\{x_1, x_2, x_3, y_1, y_2, y_3\}$ *and features a total of nine possible initial states, each represented by a set of facts* $\{x_i, y_i\}$*, where* $1 \leq i \leq 3, i \in Z$*. The initial probability function is given by:*

$$I = \begin{cases} \{x_1, y_1\} \mapsto .04 & \{x_1, y_2\} \mapsto .14 & \{x_1, y_3\} \mapsto .02 \\ \{x_2, y_1\} \mapsto .14 & \{x_2, y_2\} \mapsto .49 & \{x_2, y_3\} \mapsto .07 \\ \{x_3, y_1\} \mapsto .02 & \{x_3, y_2\} \mapsto .07 & \{x_3, y_3\} \mapsto .01 \end{cases}$$

and $I(s) = 0$ *for any other states. The problem includes four actions:* $A = \{\text{GO\_E}, \text{GO\_S}, \text{GO\_N}, \text{GO\_W}\}$*. The actions* GO_E *and* GO_W *have the precondition:* $pre(\text{GO\_E}) = y_3$ *and* $pre(\text{GO\_W}) = y_3$*, indicating the agent can only move horizontally when located in the top row. The goal is* $x_2 \wedge y_2$ *and* $\tau = 0.75$*. When* $\tau = 1$*, the problem asks for a plan that satisfies all possible initial states, which, by definition, is a CP problem. When* $\tau = 1$ *and there is only one possible initial state, by definition, it becomes a classical planning problem.*

1.1.7 CPCES

A CP problem can be viewed as a path finding problem over a directed graph (Bonet and Geffner 2000) where each vertex represents a "belief state"—possible states the system could be in—and each edge represents the effects of an action. Due to the uncertainty of the initial state, the number of belief states grows exponentially, making the problem significantly challenging. In fact, CP is ExpSpace-Complete (Haslum and Jonsson 2000; Bonet 2010). Given that it is infeasible to consider all possible belief states during the search, Grastien and Scala (2020)

Algorithm 1 The conformant planner `CPCES`.

```
1: input: conformant planning problem P
2: output: a conformant plan, or no plan
3: B := ∅                                              ▷ empty sample set
4: π := ϵ                                              ▷ empty plan
5: loop
6:     q := generate_counter_example(P, π)
7:     if q ≠ ⊥ then
8:         return π
9:     B := B ∪ {q}
10:    π := produce_candidate_plan(P, B)
11:    if π = ⊥ then
12:        return no plan
```

proposed an alternative approach using counter-examples to guide the search for solutions to CP problems, naming their algorithm `CPCES`.

`CPCES` (summarized in Algorithm 1) is a conformant planner that has been proven to be both sound and complete. It iteratively searches for candidate plans and counter-examples until no counter-example remains, at which point the final candidate plan becomes the valid solution. A counter-example (also called a "sample") is an initial state for which the candidate plan is invalid. Specifically, `CPCES` begins with an empty sample set B and an empty plan π. The planner then enters an iterative process, in which it generates a counter-example to the current candidate plan (Line 6), adds it to the sample set B (Line 14), and produces a new candidate plan (Line 15). This new candidate plan ensures that all initial states in B can eventually reach the goal state, after which π is updated with the new candidate plan. The details of `CPCES` will be introduced in Section 3.1.

`CPCES` cleverly avoids the traditional path-finding search method, which can potentially involve a large number of belief states. Instead, it focuses on searching for counter-examples and updating candidate plans. This approach not only significantly reduces memory usage but also eliminates the need to consider all possible initial states, making the search fast and efficient.

Example 6. *We use the example in Figure 1.2 to illustrate how `CPCES` solves a CP problem. The changes in variables at each step are summarized in Table 3.1.*

Table 1.1: Detailed process of how CPCES solves the problem illustrated in Figure 1.2.

Iteration	Sample	Sample Set	Candidate Plan
0	-	$\emptyset$	ε
1	(2,2)	$\{(2,2)\}$	GO_E,GO_N
2	(2,5)	$\{(2,2),(2,5)\}$	GO_N $\times$ 3, GO_E, GO_S $\times$ 2
3	(4,1)	$\{(2,2),(2,5),(4,1)\}$	GO_N $\times$ 4, GO_E $\times$ 3, GO_W $\times$ 2, GO_S $\times$ 2
4	(1,1)	$\{(2,2),(2,5),(4,1),(1,1)\}$	GO_N $\times$ 4, GO_E $\times$ 4, GO_W $\times$ 2, GO_S $\times$ 2
5	-	-	

0. *The sample set B is initially empty, and the candidate plan is $\pi_0 = \epsilon$.*
1. *Since the candidate plan is ϵ, any initial state that differs from the goal state can be a counter-example. Suppose **CPCES** selects (2,2) as the counter-example and adds it to B. **CPCES** then updates the candidate plan to $\pi_1 =$* GO_E, GO_N.
2. ***CPCES** identifies a new counter-example to π_1 at (2,5) and adds it to B. Since next candidate plan must satisfy all counter-examples in B, **CPCES** updates it to $\pi_2 =$* GO_N $\times$ 3, GO_E, GO_S $\times$ 2.
3. ***CPCES** finds a counter-example to π_2 at (4,1) and adds it to B. The candidate plan is then updated to $\pi_3 =$* GO_N $\times$ 4, GO_E $\times$ 3, GO_W $\times$ 2, GO_S $\times$ 2.
4. ***CPCES** detects a counter-example to π_3 at (1,1) and adds it to B. The candidate plan is updated to $\pi_4 =$* GO_N $\times$ 4, GO_E $\times$ 4, GO_W $\times$ 2, GO_S $\times$ 2.
5. ***CPCES** finds no counter-example exists to π_4, indicating that the final π_4 is a valid solution to the problem. **CPCES** returns π_4 and the algorithm terminates.*

1.1.8 Hitting Set

In this thesis, we use the hitting set to avoid the occurrence of disjunctions in the goal states of classical planning problems translated from PCP, thereby enabling the classical planner to solve them more efficiently.

The hitting set problem is a fundamental combinatorial optimization problem

(Karp 2009). Let B be a set of elements and $A = \{B_1, \ldots, B_n\}$ so that each $B_i \subseteq B$. The goal of hitting set problem is to find a smallest subset $H \subseteq B$ such that $H \cap B_i \neq \emptyset$ for all $B_i \in A$. In other words, the hitting set must "hit" or intersect every set in the collection. For example, given $A = \{\{a, b, c\}, \{a, d\}, \{c, d, e\}\}$, a valid hitting set is $H = \{c, d\}$, since it intersects each of the three subsets.

Computing hitting set is NP-COMPLETE (Karp 2009), even when restricted to special cases such as 3-Hitting Set, where each subset has at most three elements (Garey and Johnson 2002). A variety of methods has been created to compute hitting sets, including branch-and-bound techniques (Bläsius et al. 2022), enumeration algorithms (Gainer-Dewar and Vera-Licona 2017), Integer Linear Programming formulations (Reiter 1987), and SAT-based approaches (Biere et al. 2009).

The hitting set problem has found use in numerous applications. For example, in testing and diagnosis, hitting sets are used to minimize the number of test cases or to identify minimal fault explanations in model-based diagnosis (Reiter 1987). In information retrieval and databases, hitting sets provide a foundation for query optimization and minimal keyword selection (Kavvadias and Stavropoulos 2005).

1.2 Contributions

In 2020, Grastien and Scala introduced a pioneering approach to conformant planning based on counter-examples. Their method, `CPCES`, iteratively refines candidate plans by identifying counter-examples (initial states) for which the current plan fails and incorporating this information into the planning process until a valid plan is found. A key strength of this approach lies in its ability to avoid reasoning over the entire set of possible initial states, thereby offering significant advantages in efficiency over traditional conformant planners.

Despite the conceptual elegance and practical effectiveness of `CPCES`, our analysis reveals several limitations that constrain its overall efficiency. First, we observe that the number of predicates involved in the classical abstraction grows significantly as the number of refinement iterations increases. This, in turn, slows

down the plan refinement phase and increases computational overhead. Second, although certain initial states can rapidly guide CPCES toward a valid plan, the algorithm lacks a mechanism for identifying and prioritizing such "helpful" initial states, leading to inefficient exploration. Third, the use of the Fast Downward System (Helmert 2006) as the classical planner within CPCES, while flexible, results in substantial computational delays in practice. To address these issues, we propose three targeted enhancements to the original CPCES framework. The first, merging certain facts, reduces the complexity of the classical abstraction by identifying and combining known predicates during plan executions. The second, warm-starting CPCES, introduces a mechanism to select "important initial states" early in the search process. The third, incremental CPCES, improves the efficiency of plan searching when using Fast Downward in CPCES, by incrementally generating SAS+ file. Empirical evaluations demonstrate that these enhancements yield considerable improvements in planning efficiency.

Motivated by the success of CPCES in conformant planning, we hypothesize that the counter-example based approach can be generalized to other forms of planning under uncertainty. In this thesis, we focus on probabilistic conformant planning, which extends the conformant planning by incorporating probabilistic distribution in the initial states. Building on CPCES, we develop two novel algorithms: p-CPCES and p-CPCES-hit. Experimental comparisons indicate that p-CPCES-hit outperforms the state-of-the-art probabilistic conformant planner, Probabilistic-FF, particularly when solving complex problems.

Nevertheless, a key limitation of the CPCES-based family of algorithms is that they are inherently single-threaded, which makes them vulnerable to performance bottlenecks in any of their computational components. To overcome this challenge, we hypothesize that parallelizing the planning process could significantly improve overall efficiency. We develop a parallelized version of p-CPCES-hit by decomposing the algorithm into three independent modules and executing each module using multiple workers across available CPU cores. The resulting system, parallel-CPCES, demonstrates substantial speedups over p-CPCES-hit and probabilistic-FF. Furthermore, we observe that performance gains scale positively with the number of parallel workers, confirming the effectiveness of paral-

lelization in mitigating bottlenecks.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 1 provides the introduction, outlining the research motivation and an overview of the work. Chapter 2 provides a review of related work in the field. Chapter 3 presents three methods to improve the efficiency of the `CPCES` algorithm. Chapter 4 proposes a counter-example based approach for solving PCP problems. Chapter 5 introduces a parallel framework that leverages multi-core CPUs to solve PCP problems. Finally, Chapter 6 presents the conclusions of this thesis and discusses directions for future work.

Chapter 2

Literature Review

In this chapter, we begin by reviewing heuristic search methods that have significantly influenced the field of AI planning. We then introduce the fundamentals of classical planning, followed by an overview of the Planning Domain Definition Language (PDDL), which has become the *de facto* standard for modeling planning problems. Building on this foundation, we examine various extensions beyond classical planning, including conformant planning and probabilistic conformant planning. Finally, we present the Counter-Example Guided Abstraction Refinement (CEGAR) framework.

2.1 Heuristic Search Methods

In the field of AI planning, many mainstream planners, such as `Fast Forward` (Hoffmann 2001), `Fast Downward` (Helmert 2006), `Probabilistic-FF` (Domshlak and Hoffmann 2006, 2007), rely on search algorithms to find a valid plan that transitions the system from an initial state to one of the goal states. Search algorithms can be broadly classified into non-heuristic search and heuristic search.

- Non-heuristic search explores the search space based solely on the problem's structure and transition rules, without using any heuristic estimates (future costs estimates) to prioritize which paths to explore. Examples of non-heuristic search algorithms include Breadth-First Search (BFS), Depth-First Search (DFS), Uniform Cost Search (UCS), and Iterative Deepening Search (IDS) (Winston 1992). Figure 2.1 presents a simple example of BFS.
- Heuristic search, on the other hand, makes use of a heuristic function to

estimate the desirability of a given state, guiding the search toward more promising regions of the search space. Typical heuristic search algorithms include Best-First Search, A*, and Weighted A* (Winston 1992).

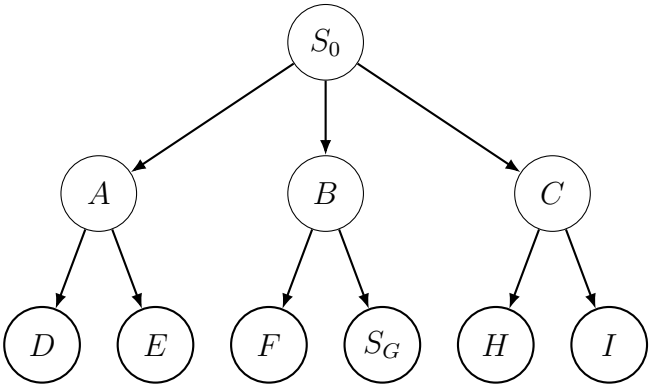


Figure 2.1: The figure shows the BFS traversal starting from the initial state S_0, expanding nodes level by level in the order: $S_0 \rightarrow A, B, C \rightarrow D, E, F, S_G, H, I$. The goal state S_G is found after expanding node B's children. No heuristic estimates are used during the search.

A heuristic function provides an estimate of the cost from a given state to a goal state. During a heuristic search process, the planner starts from the initial state and expands reachable successor states, typically computing heuristic values for each. Search strategies will prioritize expanding states based on an evaluation that combines both the cost accumulated so far and the heuristic estimate, thereby guiding the search toward more promising parts of the state space and aiming to find a plan to the goal efficiently.

A* (Hart et al. 1968) combines the real cost $g(s)$, which is the cost from the initial state S_0 to the current state s, and the estimated cost $h(s)$, which is the estimated cost from the current state to the goal (also known as the heuristic value). It uses their sum $f(s) = g(s) + h(s)$ as the evaluation function to guide its exploration of the state space. During the search process, A* maintains two lists: an open list that contains states that have been discovered (seen) but not yet fully expanded, and a closed list that contains states that have already been fully expanded. This mechanism prevents redundant exploration and enables efficient traversal of the search space. For A* to guarantee finding an optimal plan, the heuristic function $h(s)$ must be admissible, meaning it never overestimates the true minimal cost

from any state s to the goal. That is, for every node s, we must have $h(s) \leq h^*(s)$, where $h^*(s)$ is the actual cost of the cheapest path from s to the goal.

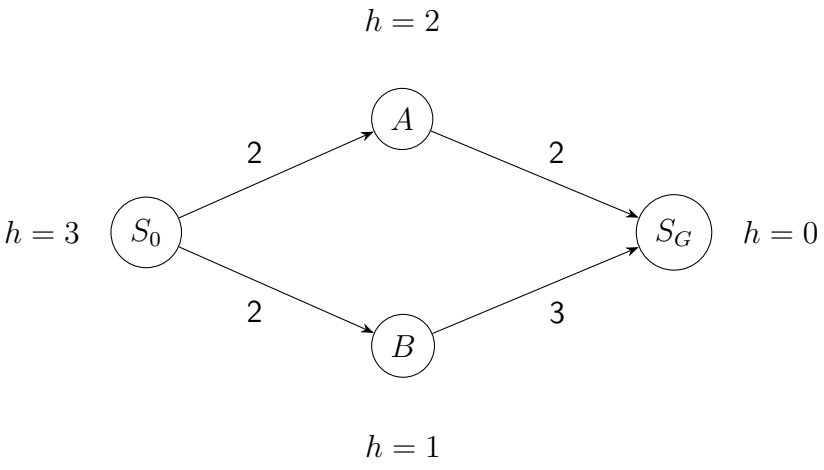


Figure 2.2: An example of A* search applied to a simple planning problem. Each node is annotated with a heuristic value h, representing the estimated cost to reach the goal. The edge labels denote action costs between states. This figure highlights how A* evaluates paths by balancing the actual cost so far with the heuristic estimate, illustrating the algorithm's preference for the path through node A (total cost = 4) over node B (total cost = 5).

Example 7. *Figure 2.2 illustrates the process of solving a planning problem using A* search. In the figure, each node represents a state, where S_0 denotes the initial state and S_G denotes the goal state. The numbers on the edges represent the cost between two states, and the h value on each node indicates the estimated cost from that state to the goal state. A* search maintains two sets of nodes: an* ***open list****, which stores states that have been discovered but not yet expanded, and a* ***closed list****, which stores states that have already been expanded.*

The A search proceeds as follows:*

1. *Initially, the open list contains only the start state:* `open` $= \{S_0\}$*, and the closed list is empty:* `closed` $= \{\}$*. The real cost $g(S_0) = 0$, the heuristic value $h(S_0) = 3$, and therefore the evaluation function $f(S_0) = 0 + 3 = 3$.*
2. *S_0 has the smallest f value in the open list, so it is expanded and moved to*

the closed list: $\texttt{open} = \{A, B\}$, $\texttt{closed} = \{S_0\}$. *- For node A:* $g(A) = 2$, $h(A) = 2$, *so* $f(A) = 4$ *- For node B:* $g(B) = 2$, $h(B) = 1$, *so* $f(B) = 3$

3. *B has the smallest f value, so it is expanded and moved to the closed list:* $\texttt{open} = \{A, S_G\}$, $\texttt{closed} = \{S_0, B\}$. *- From B, the goal node* S_G *is reached with* $g(S_G) = 5$, $h(S_G) = 0$, *so* $f(S_G) = 5$

4. *A is now the node with the smallest f value, so it is expanded:* $\texttt{open} = \{S_G\}$, $\texttt{closed} = \{S_0, B, A\}$. *- From A,* S_G *is reached again with a new path cost* $g(S_G) = 4$, $h(S_G) = 0$, *so* $f(S_G) = 4$. *Since this new path to* S_G *is cheaper than the previously discovered one (*$f = 4 < 5$*), the old entry in the open list is replaced with the better one.*

5. *Now, the open list contains only* S_G *with* $f = 4$, *so it is expanded and recognized as the goal. The search terminates. Tracing back from* S_G *through its parent nodes, the optimal path* $S_0 \to A \to S_G$ *is recovered.*

Weighted A* (Pohl 1970) relaxes the requirement for optimality in order to speed up the search. It modifies the evaluation function to $f(s) = g(s) + wh(s)$, where $w > 1$ is a weight that emphasizes the heuristic estimate. By inflating the estimated cost, Weighted A* tends to more aggressively pursue states that appear closer to the goal, thereby reducing the overall search effort but possibly sacrificing optimality. Since $wh(x)$ may overestimate the true cost to the goal even if $h(s)$ itself is admissible, the resulting evaluation function is no longer admissible. Consequently, Weighted A* does not guarantee finding the optimal solution. However, in practice, it often finds solutions of reasonably high quality while significantly reducing computation time. When the heuristic function is informative and accurate, Weighted A* provides an effective trade-off between solution quality and search efficiency.

Greedy Best-First Search (Pohl 1970) is another type of heuristic search algorithm. It always expands the node that appears most promising based on the heuristic function, using $f(s) = h(s)$. Unlike A*, it completely ignores the cost from the start state, $g(s)$, which often leads to faster search but may result in suboptimal solutions that are excessively long.

Hill Climbing (Pearl 1984) is a commonly used local search strategy in AI Planning

and is considered a type of heuristic search method. The algorithm starts from the current state and always selects the neighboring state with the lowest heuristic value $h(s)$. Unlike Greedy Best-First Search, which maintains an open list of all discovered states and expands the one with the lowest $h(s)$ globally, Hill Climbing only considers the immediate neighbors of the current state. Its data structure holds only the current state and its successors, making it faster but more prone to getting stuck in local minima. If none of the neighbors has a better heuristic value, the algorithm terminates or fails due to being trapped in a local optimum. To overcome this limitation, Enforced Hill Climbing (EHC) was introduced by Hoffmann as part of the `Fast Forward` planner (Hoffmann 2001). When no better neighbor is available, EHC performs a BFS from the current state to find a state with a strictly lower heuristic value and resumes the hill climbing from there. Compared to pure Hill Climbing, EHC is more robust and, in many practical planning domains, faster than A* – especially when optimality is not required. However, EHC does not guarantee an optimal solution.

2.2 Classical Planning

Classical planning research dates back to 1961 with the development of the General Problem Solver by Newell and Simon (1961), designed to mimic human thinking in solving simple formal problems. In 1971, Fikes and Nilsson introduced STRIPS (Fikes and Nilsson 1971), a widely accepted method for defining classical planning problems, which uses a fragment of first-order logic to represent states. Propositional STRIPS planning is PSPACE-COMPLETE (Bylander 1994a). In theory, classical planning can be viewed as a search problem: starting from an initial state, the planner systematically explores reachable states until a goal state is found, thus discovering a plan (or it has been proved none exists); however, brute-force search is highly inefficient. Consequently, researchers have developed various search algorithms to improve planning efficiency. In 1997, Blum and Furst significantly enhanced search efficiency with the `Graphplan` approach (Blum and Furst 1997). Later, in 1999, Kautz and Selman translated classical planning problems into SAT problems and integrated the `Graphplan` approach to design the Blackbox planning system, demonstrating remarkable solving efficiency (Kautz

and Selman 1999). Many planners, such as `Fast Forward` (Hoffmann 2001) and `Fast Downward` (Helmert 2006), adopt heuristic search methods to efficiently find plans.

`Fast Forward` (`FF`), created by Hoffmann in 2001, is a heuristic forward state-space planner that performs search from the initial state toward the goal state. Its primary search strategy is Enforced Hill Climbing (EHC), which combines greedy heuristic search with breadth-first search to escape local minima. If EHC fails to find a solution, `FF` switches to a complete best-first search strategy using Weighted A* (Pohl 1970) to guarantee completeness.

The heuristic used by `FF`, known as the `FF` heuristic, is based on delete-relaxation (Bonet et al. 1997), in which all negative effects of actions are ignored. This simplification transforms the original planning problem into a relaxed one, which is easier to solve. To compute the heuristic value of a state, `FF` uses the Relaxed `Graphplan` approach (Blum and Furst 1997) to expand forward from the initial state until a relaxed plan is found. The length of this relaxed plan is used as the heuristic value for the current state. In addition to estimating distance to the goal, the `FF` heuristic also identifies a set of helpful actions for each search state S (we are using the definition of CP from Definition 1). These are the actions that appear in the first step of the relaxed plan — that is, the first set of actions considered for execution, where each step corresponds to a single action — meaning they are the most promising actions to move toward the goal. The set of helpful actions $H(S)$ for a state S is then defined as:

$$H(S) := \{o \mid pre(o) \subseteq S, \mathit{eff}^+(o) \cap G_1 \neq \emptyset\}$$

where G_1 is the set of subgoals that appear at the first level of relaxed plan extraction.

During EHC and breadth-first search, `FF` restricts the successors of any state S to only those generated by the helpful actions of S, significantly reducing the search space and improving planning efficiency. However, this approach is not completeness-preserving. If EHC fails to find a solution, `FF` falls back to a complete Weighted A* (Pohl 1970) search to ensure solution discovery.

`Fast Downward` (`FD`), created by Helmert in 2006, is a heuristic search planner composed of three main components: the translator, the knowledge compilation module, and the search engine. Unlike other planners (e.g., `FF`), which directly perform plan search using the facts defined in a PDDL file, `FD` first analyzes causal relationships and identifies mutually exclusive facts, grouping them into mutex groups. Each mutex group is treated as a multi-valued variable (Bäckström and Nebel 1995a) and is written into an SAS+ file. This transformation reduces the branching factor in the state space during the search process. This step is handled by the translator. Once the SAS+ representation is obtained, the knowledge compilation module preprocesses the planning task and provides optimized data structures for `FD`. Specifically, this module first constructs a Causal Graph (CG) to analyze causal dependencies between state variables and builds Domain Transition Graphs (DTGs) to describe how variables change through actions. Based on the CG and DTGs, it computes the CG heuristic to improve search efficiency. Finally, it generates successor generators to optimize state expansion and axiom evaluators to compute derived variables. The search engine in `FD` is based on heuristic search, primarily using Greedy Best-First Search (Pohl 1970) along with several optimization techniques to improve efficiency. These optimizations include:

- Heuristic-guided search: `FD` primarily uses the causal graph heuristic and the Fast Forward heuristic to guide the search process.
- Preferred operators: Similar to helpful actions in `FF`, `FD` identifies actions that are more likely to lead to the goal and prioritizes their expansion, speeding up the search.
- Deferred heuristic evaluation: In some planning tasks, state expansion can lead to a large branching factor, making heuristic evaluation computationally expensive. To mitigate this, `FD` delays heuristic evaluation, computing the heuristic value of a successor state only when necessary, reducing unnecessary computations.
- Alternative search strategies: `FD` also supports Multi-Heuristic Best-First Search, which leverages multiple heuristics simultaneously, and Focused Iterative-Broadening Search, which controls search breadth to improve efficiency.

These optimizations allow FD to efficiently handle large-scale planning tasks, making it a competitive planner in the 4th International Planning Competition (IPC-4) (Helmert 2006). Many state-based planners have been developed based on the FD framework. One prominent example is LAMA, "Landmarks And Memory" (Richter and Westphal 2010), which introduces the concept of landmarks—important subgoals that must be achieved in every valid plan. LAMA leverages these landmarks to construct heuristic functions that guide the search more effectively. It has been fully integrated into FD and can be directly used as a built-in configuration.

Madagascar (Rintanen 2012, 2014) differs from heuristic search planners like FF and FD. Instead of heuristic search, it is a planning system based on SAT solving. Its core idea is to transform a planning problem into a SAT problem and then use a SAT solver to efficiently search the solution space. Moreover, it supports parallel SAT solving (Rintanen et al. 2006, 2004), enhancing computational efficiency and scalability to handle larger planning problems. Madagascar adopts a standard SAT encoding to represent planning problems in the form:

$$\phi_t = I \wedge T(0,1) \wedge T(1,2) \cdots \wedge T(t-1,t) \wedge G$$

where I represents the initial state, G represents the goal state, and $T(i, i+1)$ encodes the state transitions, specifying possible actions between step i and $i+1$. The satisfiability of ϕ_t indicates that a feasible plan exists within t steps. There are three core technologies used in Madagascar.

1. Implementation of ∃-Step Encoding: Unlike traditional ∀-semantics used in early SAT-based planning approaches, which enforce mutual exclusion between all actions in the same time step, Madagascar's ∃-step encoding (Rintanen et al. 2006) allows multiple conflicting actions to be planned at each step, leading to more compact encodings. This approach also eliminates unnecessary action exclusion constraints, significantly reducing the size of the SAT formula.

2. Mutex Constraints: Madagascar uses 2-literal invariants to accelerate SAT solving. These constraints help prune the search space by enforcing logical relationships between state variables.

3. Parallel SAT Solving and Scheduling Strategies: Traditional SAT-based planning methods solve satisfiability problems sequentially by incrementally increasing the time horizon t (Kautz and Selman 1999). `Madagascar` introduces two parallel scheduling strategies. Strategy A (Rintanen et al. 2006) distributes computational resources evenly across a fixed set of plan lengths, enabling parallel exploration of multiple horizons. Strategy B (Rintanen et al. 2006) employs a parallelized search strategy that allocates resources geometrically across different plan lengths. By interleaving the exploration—assigning more time to shorter horizons while still allocating resources to longer ones—it avoids getting stuck on difficult unsatisfiable cases and often identifies valid plans more efficiently.

With these optimizations, `Madagascar` demonstrates high scalability in classical planning, enabling it to handle larger and more complex planning problems while surpassing the efficiency of many earlier SAT-based planning approaches.

Figure 2.3 illustrates the simplified workflow of `Madagascar` for solving a classical planning problem. Initially, the user defines the initial step i (the starting plan length for the search) and the maximum step k (the upper bound on the plan length considered). Then, the algorithm enters an iterative process. At each iteration, `Madagascar` encodes the classical planning problem into a CNF formula corresponding to a plan of length i, and solves the formula using a SAT solver. If the SAT solver finds a satisfying assignment, a valid plan is extracted and the algorithm terminates. If the SAT solver determines that the formula is unsatisfiable, the value of i is incremented by one. If the updated i satisfies the bound $i < k$, the process repeats with the new plan length. Otherwise, if $i \geq k$, the algorithm concludes that no plan of length at most k exists for the given planning problem, and terminates.

2.3 Planning Domain Definition Language

The Planning Domain Definition Language (PDDL) has become the *de facto* standard language for specifying planning problems in Artificial Intelligence. It was first introduced in 1998 to provide a declarative problem description language for

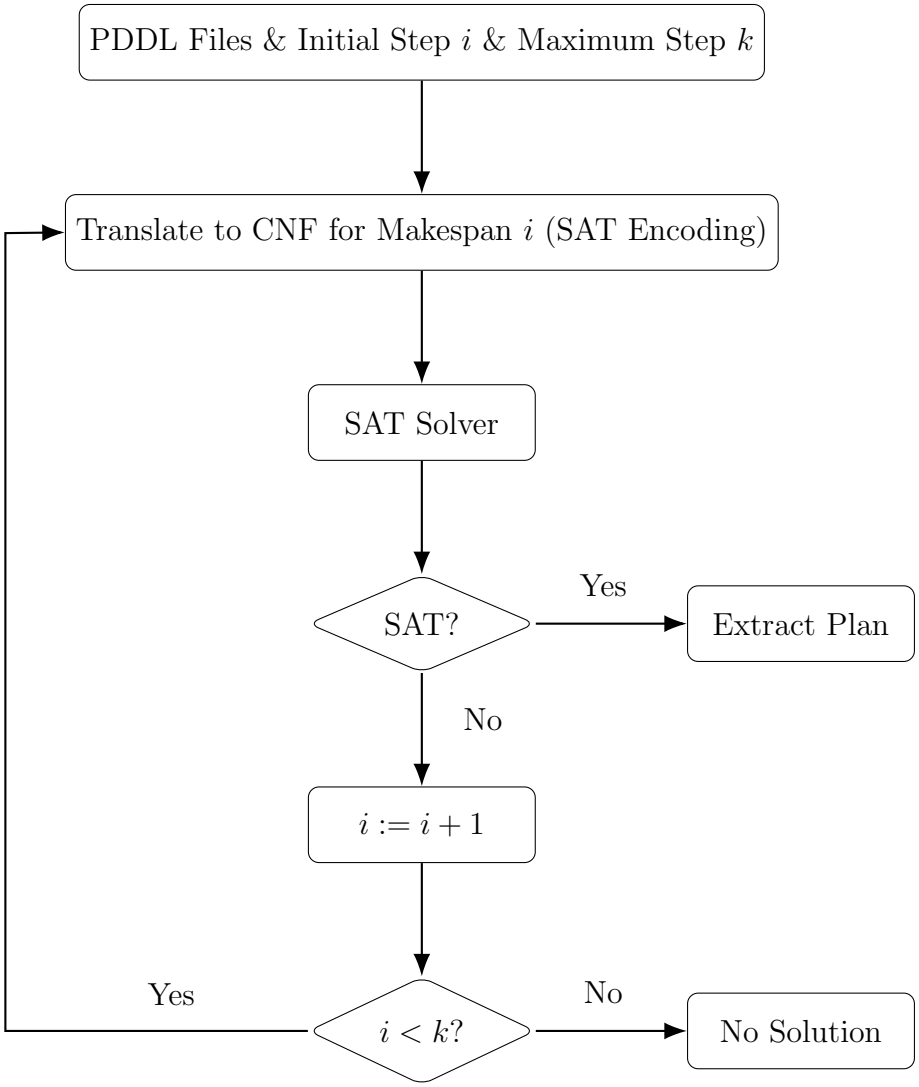


Figure 2.3: A simplified workflow of `Madagascar` for solving classical planning problems. Note that this is a simple representation of the *ramp-up* query strategy, and that the makespan increment (1 above) is the setting we use in the sequel.

the 1st International Planning Competition (IPC-1). Since its inception PDDL has undergone several significant extensions to model increasingly broad requirements facing AI agents — e.g., epistemic and aleatoric uncertainty, durational actions, continuous quantities, actions costs, axioms, etc.

The original PDDL1.0 (McDermott et al. 1998) was inspired by the STRIPS formalism and retained its core principles: planning problems are modeled using logical predicates to represent world states, and actions are defined by their preconditions and effects. PDDL1.0 facilitated the description of classical planning problems, in which the environment is fully observable, deterministic, and features no exogeneity during plan execution. This foundational version enabled researchers to share benchmark domains and directly compare planning algorithms in a standardized way. In 2003, PDDL2.1 was introduced by Fox and Long to significantly extend the expressive power of PDDL1.0, allowing engineers to model temporal and numeric domains. The main motivation behind this revision was to bridge the gap between academic planning benchmarks and real-world applications that involve time constraints, resource consumption, and continuous world dynamics. PDDL2.1 added several critical features, including : numerical fluents to model resources, durative actions to model continuous time and action concurrency, and metrics to model a broad range of planning objectives. Following these enhancements, to support the 4th International Planning Competition (IPC-4) in 2004, Edelkamp and Hoffmann proposed PDDL2.2, which introduced two notable features: derived predicates (a.k.a. axioms), and timed initial literals to model deadlines. Changes made to the language at this time maintained backward compatibility with earlier specifications of PDDL. In 2005, Gerevini and Long extended the language to PDDL3.0 to further support plan quality modeling. The language extensions in this case enabled engineers to specify the desirability of plans, and not just properties of plan feasibility and optimality. Users were able to specify (in modal temporal logic) conditions a plan should satisfy, preferences relating to the desirability of goal conditions, and numerical penalties for constraint violation. In 2008, PDDL3.1 (Geffner 2000) introduced *function symbols*, allowing fluents—function terms whose values vary across states—to return objects rather than being limited to Boolean or numeric values, thereby enhancing

the expressiveness of the language for modeling complex planning problems.

Listing 2.1: PDDL problem domain.

```
(define (domain simple-move)
  (:requirements :strips :typing)
  (:types robot location)

  (:predicates
    (at ?r - robot ?l - location)
    (left-of ?l1 - location ?l2 - location)
  )

  (:action move-right
    :parameters (?r - robot ?l1 - location ?l2 - location)
    :precondition (and
      (at ?r ?l1)
      (left-of ?l1 ?l2)
    )
    :effect (and
      (not (at ?r ?l1))
      (at ?r ?l2)
    )
  )
)
```

Listing 2.2: PDDL problem instance.

```
(define (problem move-two-locations)
  (:domain simple-move)
  (:objects
    robot1 - robot
    loc1 loc2 - location
  )
  (:init
    (at robot1 loc1)
    (left-of loc1 loc2)
```

```
  )
  (:goal
    (at robot1 loc2)
  )
)
```

Listings 2.3 and 2.3 present a simple classical planning problem in PDDL. In Listing 2.3, an action `move-right` is defined, while Listing 2.3 specifies the objects, initial state, and goal state. In this problem, there is a robot (`robot1`) and two locations (`loc1` and `loc2`). The robot is only capable of performing one action: `move-right`. The initial state specifies that the robot is located at `loc1`, and `loc1` is to the left of `loc2`. The goal state is for the robot to be located at `loc2`.

In parallel with the deterministic extensions, Younes and Littman (2004) developed PPDDL1.0, a probabilistic extension of PDDL2.1 used for the probabilistic track of IPC-4. PPDDL enables the specification of planning problems featuring quantitative uncertainty, where actions may have stochastic effects and the initial state may be partially unknown. The semantics of PPDDL are formally defined in terms of Markov Decision Processes (MDPs), making it suitable for decision-theoretic planning. By supporting nested probabilistic and conditional effects, PPDDL provides a highly expressive framework for modeling stochastic domains, while remaining compatible with classical planning constructs.

In this thesis, the conformant planning problems are taken from the benchmarks of the International Planning Competition 2008 (Geffner 2000). While the standard PDDL specifications (PDDL1.0 – PDDL3.1) do not provide constructs for modeling uncertainty in the initial state, conformant planning systems such as Conformant-FF introduced the non-standard oneof construct (Geffner 2000). This extension allows representing disjunctive information in initial states. The probabilistic conformant planning problems are adapted from these conformant planning problems and are represented using the PPDDL (Younes and Littman 2004).

2.4 Conformant Planning

Conformant Planning (CP) has attracted considerable research attention since it was first defined by Smith and Weld in 1998 who applied the graph plan method—developing separate plan graphs for each possible world—to solve CP problems. Two years later, Bonet and Geffner (2000) adapted heuristic search methods from path-finding algorithms to solve planning problems with incomplete information. A belief state represents a set of possible world states consistent with the agent's knowledge. Due to the astronomical number of such belief states — $O(2^{2^n})$, where n is the number of propositions — the path-finding approach is not well-suited for CP. In fact, the complexity of CP is ExpSpace-Complete (Haslum and Jonsson 2000; Bonet 2010). Compared to classical planning, which can be solved using path-finding methods (Hoffmann 2001), CP is far more challenging and computationally expensive. As a result, researchers have sought alternative methods that avoid expanding belief state nodes to search for solutions. Three prominent approaches have emerged for solving CP.

The first approach involves compacting the representation of belief states. In the path-finding method, belief states are combined into single nodes, significantly reducing the number of nodes that need to be expanded, thereby increasing the efficiency of solving CP problems. To et al. (2010) represented belief states using a specialized type of CNF formula and developed a technique called one-of relaxation. Bertoli et al. 2001 used a tight integration of symbolic techniques, combining BDD, "Binary Decision Diagrams", (Bryant 1986) and heuristic search to efficiently explore plans. Similarly, Darwiche and Marquis (2002a), Huang and Darwiche (2007), Cimatti and Roveri (2000), and Cimatti et al. (2004) used BDDs to compactly represent belief states, further enhancing the efficiency of CP. A key challenge of these approaches is devising effective methods to target the specific structure of the problem at hand. To address this limitation, some researchers have proposed a more explicit exploration of belief states that directly uses the power of heuristic search (Bonet and Geffner 2000). A notable contribution in this area comes from Hoffmann and Brafman (2006) , who developed the `Conformant-FF` planner, an extension of `FF` (Hoffmann 2001). In `Conformant-FF`,

each search node in the search space is represented as a tuple:

$$S = (I, action\_sequence\_act)$$

where I is the initial belief state, represented as a CNF formula, which defines all possible initial world states, and *action_sequence_act* is the sequence of executed actions, representing the actions applied from the initial state. `Conformant-FF` does not explicitly store the complete belief state set (i.e., all possible worlds). Instead, it only keeps the initial belief state and the executed action sequence. This "lazy representation" avoids the need to store large belief state sets, significantly reducing memory overhead. However, as a consequence, `Conformant-FF` must infer the current belief state through logical reasoning rather than direct state enumeration. It employs SAT-based reasoning to determine which predicates are known in the current belief state. In the search procedure, `Conformant-FF` performs heuristic forward search in belief space, using the relaxed planning heuristic from `FF` (Hoffmann 2001). When expanding a belief state, `Conformant-FF` considers all applicable actions whose preconditions hold in all possible worlds. For each action, it evaluates all conditional effects whose conditions are guaranteed to be **True** in all possible initial states, and applies their effects to compute the successor belief state. This process continues until a belief state satisfying the goal is reached.

The second approach involves translating a CP problem into a classical planning problem. One of the most well-known planners that applies this method was developed by Palacios and Geffner (2009). Their approach introduces mapped literals K_L/t, where L represents (sub)goal literals, and t, called a *tag*, is a set of literals encoding a partial assumption about the initial state. The expression K_L/t denotes that L is known to be **True** if the initial assumptions t hold. In the simplest translation $K_0(P)$, only the empty tag t_0 is used, corresponding to knowledge that does not rely on assumptions. The transformation process systematically eliminates uncertainty through support and cancellation rules, ensuring that knowledge is correctly propagated and negative inferences are properly managed. This transformation converts a CP problem P into a classical planning problem $K_0(P)$, which can then be solved using off-the-shelf classical planners. The translation process con-

sists of: (i) Fluents: new literals K_L/t representing conditions under which goals hold **True**; and (ii) Actions: each CP action is mapped into a classical action with effects conditioned on prior knowledge. To ensure correct knowledge propagation, two types of rules are introduced: (i) Support Rules, which enforce that if a condition C is known to be **True**, then its effect L is also known ($a : K_C/t \rightarrow K_L/t$); and (ii) Cancellation Rules, which remove negative knowledge when uncertainty about a precondition persists ($a : \neg K_{\neg C}/t \rightarrow \neg K_{\neg L}/t$). Their method, implemented in the `T0` planner (Bonet et al. 2009), was the top-performing system in the 2006 International Planning Competition (IPC-6) Conformant Track (Palacios and Geffner 2009), demonstrating superior scalability compared to belief-space search methods. This compilation-based approach has become a foundational technique in CP research, as it effectively reduces the complexity of handling uncertainty while maintaining soundness and completeness in problems with bounded conformant width — The conformant width of a problem, analogous to the width in graphical models (Dechter 2003), provides an upper bound on the time and space complexity required to construct such translations: the construction is exponential in this width, which is always less than or equal to the number of fluents, but often much smaller in practice.

The third approach employs a sampling-based mechanism, similar to `CPCES`. An example of this approach is the `Fragment` Planner (Kurien et al. 2002). The key principle of this planner is that if a conformant plan exists for a set of initial states, then a plan must exist for each individual state, which can be incrementally extended into a conformant plan. This approach first constructs a plan for a single possible initial state and then progressively extends it to ensure it works across all possible worlds. The planner supports various search strategies, including complete and incomplete methods, as well as systematic and randomized approaches. While these strategies do not impact the planner's completeness, they significantly influence efficiency. A key advantage of Fragment-Based Planning is its ability to scale efficiently, even as the number of possible worlds grows. In 2012, a Generate-And-Complete approach (Nguyen et al. 2012) was proposed as an alternative to the `Fragment` Planner. While both approaches start by generating a potential plan for a sub-problem, Generate-And-Complete differs by maintaining a single evolv-

ing plan rather than constructing solutions from fragments. In this approach, an initial plan is first generated for one possible initial state using a classical planner. Then, in a completion phase, the plan is progressively modified and extended by inserting necessary actions to ensure its applicability to other possible initial states. Once all sub-problems have been addressed, the final plan is checked for validity as a conformant plan. If it fails, a new possible plan is generated for the first sub-problem, and the process repeats.

2.5 Probabilistic Conformant Planning

The PCP problem was first defined by Kushmerick et al. (1995). Their planner, `Burdian`, starts by computing an initial plan that satisfies some subgoals. The plan is then modified and additional actions are inserted to ensure that it satisfies more initial states, continuing this process until the desired probability threshold is reached. Like CP problems, PCP problems can theoretically be solved using a path-finding approach, expanding and tracking belief states until the probability of reaching the goal state meets the required threshold. However, there are two key challenges in solving PCP problems: (i) finding a plan, and (ii) calculating the probability of success for that plan. Since the path-finding approach is inadequate even for CP problems, it is even less suitable for PCP problems, as the addition of probability distributions for initial states and a probability threshold for goal states makes the problem significantly more complex. Some well-known methods have been developed to tackle PCP problems.

`Probapop` (Onder et al. 2006) is a conformant probabilistic planner designed to handle uncertain initial states and uncertain action effects. Its core strategy is to implement the `Buridan` probabilistic planning algorithm (Kushmerick et al. 1995) on top of `Vhpop` (Younes and Simmons 2003). The `Buridan` planning algorithm is based on Partial-Order Planning (`POP`), supporting actions with probabilistic effects and utilizing an incremental optimization strategy to improve the probability of success. `Vhpop` is a heuristic partial-order planner that incorporates Flaw Selection Strategies and heuristic search, making it more efficient than traditional `POP` planners for large-scale problems.

Another approach involves translating a PCP problem into a classical planning problem. Taig and Brafman (2012; 2013) proposed a translation-based method inspired by the conformant planner developed by Palacios and Geffner (2009). In 2012, Taig and Brafman proposed two compilation methods: Resource-Constrained Classical Planning (RCCP) and Cost-Optimal Classical Planning (COCP). In RCCP, a numeric resource variable is introduced to model the total probability mass of initial states that the planner is allowed to ignore. To ensure that the resulting plan achieves the goal with a probability of at least τ, the planner is permitted to ignore only those initial states whose combined probability does not exceed $1 - \tau$. This is operationalized through special "ignore actions", where each action corresponds to a *tag* — i.e., a subset of initial states sharing similar uncertainty — and consumes an amount of the resource equal to the probability mass of the corresponding states. The total resource is initialized to $1 - \tau$, ensuring that ignored states cannot collectively exceed this bound. Consequently, the planner must produce a conformant plan that succeeds for the remaining, non-ignored initial states whose total probability is at least τ. Their transformation process consists of five main steps:

1. Each possible initial state is assigned a tag, representing a particular combination of uncertainty (e.g., truth values of certain fluents). These tags are disjoint, complete, and deterministic.
2. The original PCP actions are transformed to operate over tag-indexed propositions p/t, where p/t indicates that p holds in all initial states covered by tag t.
3. The planner can execute "ignore actions" to ignore low-probability tags. Each "ignore action" consumes a portion of a limited numeric resource equal to the probability of the tag. The sum of these costs must not exceed $1 - \tau$.
4. Once a tag t is dropped, the planner is allowed to introduce assumptions about the truth value of any proposition under that tag. This is done via "assume actions", which add p/t to the current belief state, enabling further reasoning even without evidence.
5. When a proposition p is known to be **True** under all remaining (i.e., non-dropped) tags, a "merge action" is used to infer that p is globally **True**.

Formally, a "merge action" has all relevant p/t as preconditions and adds p as its effect.

After applying these five steps, the original PCP problem is transformed into a classical planning problem. Experimental results show that RCCP is computationally efficient but may yield suboptimal plans (since it stops once a plan with success probability τ is found). In contrast to RCCP, the COCP compilation aims not only to find a plan that achieves the goal with a success probability of at least τ, but to maximize the success probability by minimizing the total probability of the ignored initial states. COCP uses a similar compilation structure as RCCP. However, rather than enforcing a hard constraint on the total probability mass of dropped tags (i.e., a resource limit of $1-\tau$), COCP treats this probability mass as the cost to be minimized. Specifically, each "ignore action" is given a cost equal to the probability of the tag. All other actions, including "merge actions" and "assume actions", are assigned zero cost. Experimental results show that COCP produces plans with maximal probability of success but requires greater computation time. Building upon this work, in 2013, Taig and Brafman refined the Compilation Approach by introducing Cost-Bounded Classical Planning (CBCP), which improves the efficiency of solving PCP problems within a cost constraint (Taig and Brafman 2013). CBCP retains the same five-step transformation process but introduces cost-bounded planning as a key improvement. Instead of treating ignored states as a constrained resource (like in RCCP), CBCP explicitly incorporates cost bounds into the planning process. The cost function represents the total probability of ignored states, ensuring that the planner finds a feasible solution within a predefined cost threshold. This approach offers greater flexibility and performs better in some domains compared to RCCP and COCP.

The third planner, `CPplan`, converts a PCP problem into a CSP/SAT problem. It was introduced by Hyafil and Bacchus in 2003 and further refined in 2004. In `CPplan` (Hyafil and Bacchus 2003), each state is represented by multiple variables, and state transitions are modeled using decision trees, which capture probabilistic effects of actions. When encoding the problem as a CSP, each timestamp includes three groups of variables:

- State variables: Represent the system's state at that time step.

- Action variables: Represent the action taken at that time step.
- Random variables: Represent the stochastic effects of the action.

`CPplan` uses a CSP solver to search for all possible action sequences and compute their probability of achieving the goal. To improve efficiency, the authors later introduced Algebraic Decision Diagrams (ADDs) to store and compute probability distributions during planning, replacing the earlier tabular representation (Hyafil and Bacchus 2004). By compactly representing functions over state-plan pairs, ADDs enable the reuse of identical computations across different states and plan suffixes. This structural sharing significantly reduces redundant storage and computation, thereby lowering memory usage and enhancing the algorithm's scalability.

The fourth approach, `POND`, was created by Bryce et al. in 2006b. In classical planning, heuristic search is more effective than CSP/SAT-based methods. However, in probabilistic planning, existing approaches (such as CSP/SAT encodings) struggle to scale to large problems, and the effectiveness of heuristic search is limited by the lack of efficient reachability heuristics. The authors proposed a new approach that combines planning graph heuristics with Sequential Monte Carlo (SMC) methods (Doucet et al. 2001) for PCP to approximate reachability heuristics. The Monte Carlo Labeled Uncertainty Graph (McLUG) integrates Monte Carlo sampling with the Labeled Uncertainty Graph (Bryce et al. 2006a) to compactly represent possible worlds and compute relaxed plans as heuristic guidance for search. Finally, the approach employs Weighted A* search, leveraging the heuristic computed by McLUG, to efficiently find a valid plan. This significantly improves search efficiency while maintaining reasonable plan quality.

The fifth method is `Probabilistic-FF` (`P-FF`), a heuristic forward search approach that extends `Conformant-FF` to probabilistic planning (PCP) scenarios (Domshlak and Hoffmann 2006, 2007). It can handle PCP problems with both initial state uncertainty and action effect uncertainty. `P-FF` uses Time-Stamped Bayesian Networks (TSBNs) to implicitly represent belief states and capture probabilistic dependencies in state transitions.

Example 8. *Figure 2.4 illustrates an example of how to construct TSBNs. In this example, the map consists of two rooms: Room1 (R1) and Room2 (R2). Initially,*

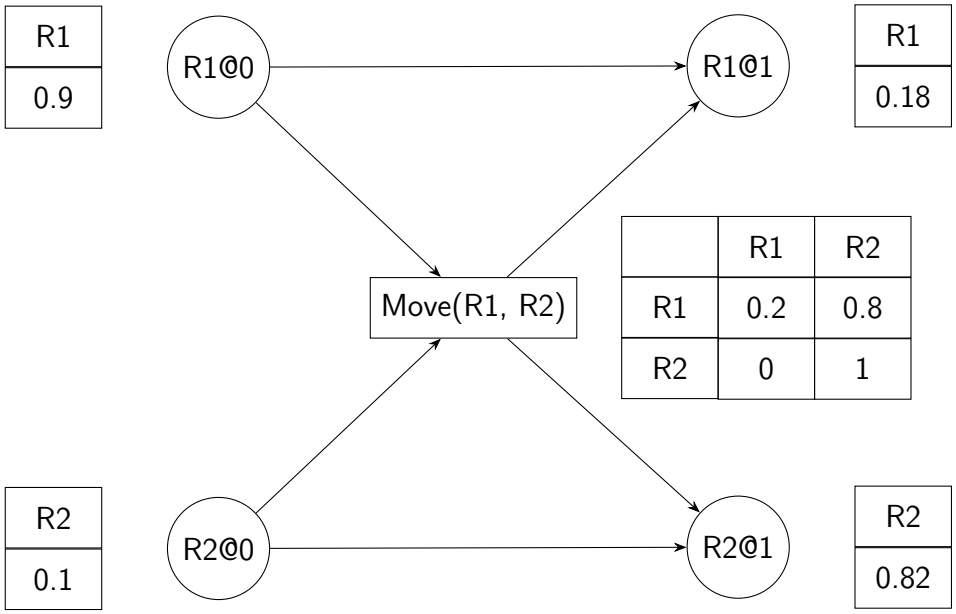


Figure 2.4: An example of Time-Stamped Bayesian Network associate with `Probabilistic-FF`. In this example the network has state nodes (circles) and action nodes (squares). The current state probability function is given by the tables adjacent each @0 state node. For example, there is 90% chance that the robot is in R(oom)1. Here, @1 nodes describe the successor state. The state transition probability information is given by the table adjacent the action nodes, here labeled "Move". For example, the probability of a transition from location R1 to R2 is 0.8.

the robot is located in Room1, but there is some uncertainty: there is a 90% *probability that it is in R1 and a* 10% *probability that it is in R2. The robot can perform an action Move(R1, R2). If the robot is in Room1, executing Move has an* 80% *chance of succeeding (moving to R2) and a* 20% *chance of failing (remaining in R1). If the robot is in Room2, executing Move(R1, R2) has no effect, and the robot stays in R2. We construct the corresponding TSBNs based on this problem (showing only one action execution). We can calculate the probability that the robot is in R2 after executing Move(R1, R2) as follows:*

- *The robot initially starts in R1 and successfully moves to R2:* $0.9 \times 0.8 = 0.72$.
- *The robot initially starts in R2, and the Move action has no effect, so it remains in R2:* 0.1.
- *Therefore, the total probability of being in R2 after the action is* $0.72 + 0.1 = 0.82$.

Similarly, the probability that the robot remains in R1 after executing Move(R1, R2) is 0.18.

It then converts states, actions, and transition rules into weighted CNF formulas and utilizes Weighted Model Counting (WMC) to compute the probability of reaching the target state, thereby avoiding the computational overhead of explicitly enumerating all possible paths. During the search process, `P-FF` applies an extended heuristic forward search to generate a planning solution while ensuring that the probability of achieving the goal state meets the predefined threshold τ. Experimental results show that `P-FF` is an order of magnitude faster than `POND` in most test domains. However, in some complex problems (such as the `LOGISTICS` domain), neither method scales well, highlighting unresolved challenges, such as how to better determine the number of times probabilistic actions should be executed to achieve a sufficiently high goal probability.

In 2006, Huang introduced a novel approach that combines knowledge compilation and search to solve PCP problems. This method encodes the PCP problem into a propositional formula, where chance variables represent uncertainty in the initial state and action effects. The formula is then compiled into a Deterministic Decomposable Negation Normal Form (d-DNNF), which enables efficient computation of

the probability that a plan will succeed. During the search process, the algorithm applies a depth-first branch-and-bound search and computes upper bounds on the success probability of partial plans at each node in the search tree. These bounds are obtained through efficient evaluation of the d-DNNF representation, allowing the algorithm to prune unpromising branches early and focus on finding an optimal plan efficiently.

2.6 Counter-Example Guided Abstraction Refinemen

`CPCES` was characterized as an implementation of CEGAR, "Counter-Example Guided Abstraction Refinement" (Grastien and Scala 2020; Clarke et al. 2000). Here, we discuss the differences and similarities between `CPCES` and CEGAR.

Model checking is an automated verification technique used to check whether a finite-state system satisfies a given logical specification (Clarke 1997). It is widely applied in hardware verification (Clarke et al. 2009; Edwards et al. 2001; Finkbeiner et al. 2015), software verification (Holzmann and Joshi 2004; Bérard et al. 2013; Jhala and Majumdar 2009), and protocol analysis (Armando and Compagna 2008; Basin et al. 2018; Lowe 1999), mainly addressing the question of whether a system's design conforms to its specifications. There are two important notions in Model Checking, which are *state explosion problem* and *abstraction*. The number of states of a system grows exponentially with the size of the system. This leads to computational intractability, making verification infeasible in practice. This is known as the "state explosion problem". Abstraction is a widely used technique to mitigate the state explosion problem by removing irrelevant details and preserving only the information relevant to the properties being verified. While early abstraction methods often relied on manual construction and domain expertise, significant progress has been made in developing automated abstraction techniques. CEGAR can be understood as a way to find proper abstractions, and thereby executions, automatically.

In Model Checking common abstraction techniques include *existential abstraction* (Clarke et al. 1994), *predicate abstraction* (Ball et al. 2001), and *data abstrac-*

tion (Derrick and Wehrheim 2007). Existential abstraction constructs a simplified model by grouping multiple concrete states into a single abstract state, ensuring (i) If the abstract model satisfies a given property, the original system also satisfies it (Soundness); (ii) If the abstract model violates a property, the original system may still satisfy it, as abstraction may introduce nonexistent behaviors (Possible Spurious Counter-Examples). Predicate abstraction represents the system using Boolean predicates rather than its original variables. Each Boolean predicate represents a logical condition and maps the original state to a combination of values of these predicates. Data abstraction is applied when a system contains numerical variables, replacing exact values with coarse-grained categories.

Example 9. *Suppose we want to verify the property that the traffic light system never shows green lights in both directions (North–South and East–West) simultaneously, to ensure safe intersection control.*

Consider a traffic light control model with the following variables:

- *Light state for each direction (Red, Yellow, Green)*
- *Timer (0–10 seconds for each light cycle)*
- *Pedestrian request button (True/False)*
- *Traffic congestion (Low, Medium, High)*

Since this verification query only concerns the mutual exclusion between light states in different directions, variables such as timer values, pedestrian requests, and congestion levels do not directly affect the property. Therefore, existential abstraction can be applied to simplify the model by removing these irrelevant variables, yielding a reduced model where only the light transitions are retained:

$$Red \rightarrow Green \rightarrow Yellow \rightarrow Red$$

By reducing the number of variables, the abstract model becomes significantly smaller, which improves the efficiency of model checking. This example is illustrative and not drawn from a specific paper.

CEGAR is an iterative refinement process that starts with an initial abstract model and refines it — i.e., incrementally adds detail to the model to eliminate

Algorithm 2 The algorithm of CEGAR.

```
1: input: A concrete model P
2: output: A refined abstract model or an execution
3: P' := initial_abstraction(P)
4: loop
5:     q := model_checking(P')
6:     if there is no such q then
7:         return P'
8:     s := check_spurious(P, q)
9:     if s = True then
10:        P' := refine_abstraction(P')
11:    else
12:        return q
```

spurious implementations (implementations that appear possible in the abstract model but are not actually possible in the concrete model) — based on counter-examples (executions that violate a desired property in the abstract model), until a precise and sufficient abstraction is found. The CEGAR algorithm is presented in Algorithm 2. Given a concrete model P, CEGAR aims to compute a refined abstraction while ensuring that the property under verification holds. The process begins with a trivial abstraction to derive an abstract model P' (Line 3). CEGAR then enters an iterative refinement loop. In each iteration, model checking is performed on P' (Line 5) to determine whether a counter-example q exists. If no counter-example is found, P' satisfies the property, and CEGAR terminates, returning P' as the verified abstraction (Line 7). On the contrary, if a counter-example is found, CEGAR checks whether it is spurious.

- If the counter-example is real ($s =$ **False** in Line 8), it indicates a genuine system bug, and the process returns this execution.
- If the counter-example is spurious (an execution that violates a property in original model but works in abstract model) ($s =$ **True** in Line 8), it suggests that the abstraction is too coarse, allowing behaviors that do not exist in the original model. In this case, CEGAR refines the abstraction by introducing additional distinctions, eliminating the spurious design (Line 10).

In addition to `CPCES`, Seipp and Helmert introduced the CEGAR framework into

classical planning and proposed a novel method for computing admissible heuristic functions (Seipp and Helmert 2018). The key idea is to iteratively refine an abstraction of the planning problem based on spurious counter-examples, so that the resulting heuristic values become increasingly accurate over time. The method begins with a trivial abstraction, typically a single abstract state that distinguishes only the initial and goal conditions without encoding any variable or operator details. In this abstraction, an abstract plan is computed — a shortest path from the initial to the goal state — which provides an estimated cost (i.e., the heuristic value). This abstract plan is then checked against the concrete planning model:

- If it corresponds to a valid plan in the original problem (i.e., all preconditions are satisfied), the heuristic value is considered admissible and can be returned.
- If not, the plan is classified as a flaw (a plan that is valid in an abstract model, but cannot be executed in the original planning problem due to missing or violated conditions), and a counter-example is extracted to explain why the plan fails.

The abstraction is then refined to eliminate this spurious behavior, typically by adding distinctions in state variables or introducing new abstract dimensions. This process is repeated iteratively, and in each iteration, the heuristic becomes more precise. Since all abstract models are over-approximations of the concrete planning problem, the computed heuristic remains admissible throughout. The process can be stopped at any time, making it suitable for anytime search strategies.

We identify five similarities between `CPCES` and CEGAR:

1. When a counter-example is found, both algorithms analyze its cause and refine the model (or plan) to ensure that the same counter-example does not appear in subsequent iterations.
2. Both methods iteratively refine their respective representations: CEGAR refines an abstract model in model checking, while `CPCES` refines classical planning PDDL file by adding initial states and modifying goal states.
3. Both methods may find a plan. In CEGAR, the returned trace exhibiting

the bug is analogous to a plan; while in CPCES, if the problem is solvable, it returns a valid conformant plan. If there is no plan in the abstract model (classical planning PDDL file), then there is no plan in the concrete model (conformant planning PDDL file).

4. In both methods, the abstract model (CEGAR) or candidate plan (CPCES) serves as over-approximations: in CEGAR, if a safety property holds in the abstract model, it is guaranteed to hold in the concrete system; similarly, in CPCES, if there is no plan for an abstract model (classical planning PDDL file), it is guaranteed that there is no plan in the concrete model (conformant planning PDDL file).
5. In CEGAR, finding the coarsest refinement is NP-hard (Clarke et al. 2000), while in CPCES, finding counter-examples is NP-hard (Grastien and Scala 2020).

We also conclude two important differences between CPCES and CEGAR:

1. In CEGAR, the abstraction for a program is an abstracted model, but in CPCES, the abstraction leads to a completely different class of problems: a conformant planning problem is abstracted into a classical planning problem.
2. In CEGAR, the counter-example to an abstracted model is an execution, while in CPCES, the counter-example is an initial state.

2.7 Conclusion

In this chapter, we conducted a systematic review of related work in AI planning. We began with an overview of heuristic search methods, with particular attention to A* and Weighted A*, which form the foundation of many planning systems. We then reviewed classical planning, outlining both its theoretical basis and representative planners. In particular, we introduced FF, FD, and Madagascar, as these planners are used in our algorithms developed in this thesis.

Next, we reviewed PDDL, which has become the standard for problem specification in AI planning, and then we traced its evolution across different versions. Next, we turned to conformant planning (CP), presented key approaches such

as compacting belief state representations, translating CP into classical planning, and sampling-based methods. Building on this, we discussed probabilistic conformant planning (PCP), highlighting methods including translations into classical or SAT/CSP problems, combinations of planning graph heuristics with statistical model checking, and heuristic forward search.

Finally, we presented the Counter-Example Guided Abstraction Refinement (CEGAR) framework, which is closely related to `CPCES`. We compared CEGAR and CPCES, clarifying both their methodological similarities and their differences.

Chapter 3

Accelerating CPCES

This chapter presents the three improvements we have made to CPCES. The contributions discussed in this chapter have been published in the proceedings of SoCS-23 (Zhang and Grastien 2023). Additionally, this chapter briefly introduces another improvement to CPCES, referred to as SUPERB, which we developed previously. The work on SUPERB has been published in the proceedings of AAAI-2020 (Zhang et al. 2020). We will use the definition of CP from Definition 2. We will use the example from Chapter 1 Section 1.1.3 as the running example throughout this chapter.

3.1 CPCES and SUPERB Algorithms

The basic structure of CPCES has been discussed in Section 1.1.7. Here, we will take a deeper look. In CPCES and SUPERB, a counter-example is an initial state that the candidate plan does not satisfy; it is also called a *sample*. The CPCES system consists of two core components: the candidate plan searching part and the counter-example searching part. We will introduce these two parts in the following subsections.

3.1.1 Computing Candidate Plans

In the candidate plan searching part, CPCES reduces CP to classical planning by introducing interpretation numbers into PDDL files. Specifically, each interpretation number corresponds to a sample from the set B (a set that stores counter-examples identified in previous iterations, defined in Line 3 of Algorithm 1). For each fact

$f \in F$, a set of cloned facts $\{(f\, int1), (f\, int2), \ldots, (f\, intn)\}$ is introduced, where each cloned fact $(f\, inti)$ represents the value of fact f, assuming the initial state is the sample s_i. The definitions of actions, the initial condition, and the goal condition are updated accordingly. This reduction yields a classical planning problem that can be solved using any classical planner.

Example 10. *Let us use the running example to explain. If sample set B contains two samples $\{s_1, s_2\}$ where $s_1 = \{x_4, y_5\}$ and $s_2 = \{x_3, y_5\}$, then in the classical planning problem, two interpretations $int1$ and $int2$ are introduced, each corresponding to a sample. For instance, the two clones of x_1 are $(x_1\, int1)$ and $(x_1\, int2)$. In the classical planning problem, the initial state is defined as*

$$\{(x_4\, int1), (y_5\, int1), (x_3\, int2), (y_5\, int2)\},$$

while the goal state is

$$\{(x_3\, int1), (y_3\, int1), (x_3\, int2), (y_3\, int2)\}.$$

Simultaneously, in the domain file, all facts in the action definitions are extended with an int parameter, and `forall` *is used to ensure that the definition applies to all clones. For example, the conditional effect of action* GO_E*, which originally states that the robot executing the action at x_1 results in the effect:*

$$(when\, (x_1)\, (and\, (x_2)\, (not\, (x_1))))$$

is modified to:

$$(\texttt{forall}\, (?int - inter)\, (when\, (x_1\, ?int)\, (and\, (x_2\, ?int)\, (not\, (x_1\, ?int)))))$$

If π is a candidate plan that satisfies two samples in B, then by applying π to both robots (i.e. one agent for each sample in the PDDL), they will execute the same actions in parallel and ultimately both of them will reach the goal state $\{x_3, y_3\}$. The classical planner used in `CPCES` *(Grastien and Scala 2020) was* `Fast Forward` *(Hoffmann 2001).*

3.1.2 Computing Counter-Examples

A counter-example is an initial state q_0 such that, when the plan $\pi = a_1 \dots a_k$ is executed (yielding a sequence of states $q_1, \dots, q_k$), the execution either fails at some step (i.e., one of the action preconditions is not satisfied) or reaches a final state that does not satisfy the goal. To detect such counter-examples, we encode the planning problem into logic: we introduce propositional variables representing the states q_i, constraints describing how these states evolve under the actions a_i, and a final constraint that enforces the failure condition above. A satisfying assignment to this encoding corresponds exactly to a counter-example.

We follow the method from Grastien and Scala (2020, Section 8.2) to compute counter-examples. The only difference is representational: Grastien and Scala use multi-valued variables to encode states, whereas we use sets of propositional facts (Boolean variables). The two encodings are isomorphic by interpreting each fact f with domain $\{0, 1\}$.

SMT variables:

- $f@i$: for each fact $f \in F$ and each time step $i \in \{0, \dots, k\}$, indicating whether f holds at step i.
- $\mathrm{Add}(f)@i$ and $\mathrm{Del}(f)@i$: Boolean formulas describing when f is added or deleted by the ith action.
- Ψ_{fail}: Boolean formula stating that the plan fails.

Constraints: The overall encoding is

$$\Phi_{P,\pi} = I[F@0] \ \wedge \ \Big(\bigwedge_{i=1}^{k} \Phi[F@(i-1), F@i] \Big) \ \wedge \ \Psi_{\mathsf{fail}}.$$

- *Initial-state constraint:* $I[F@0]$ encodes the initial condition with oneof groups (mutually exclusive facts) interpreted as exactly-one constraints.
- *Transition constraints:* for each $f \in F$ and $i = 1, \dots, k$,

$$\phi(f, i): \quad f@i \leftrightarrow \big(\mathrm{Add}(f)@(i-1) \vee (f@(i-1) \wedge \neg\, \mathrm{Del}(f)@(i-1))\big),$$

  and $\Phi[F@(i-1), F@i] = \bigwedge_{f \in F} \phi(f, i)$.

- *Failure constraint:*

$$\Psi_{\mathsf{fail}} = \Big(\bigvee_{i=1}^{k} \neg\, pre(a_i)[F@(i-1)] \Big) \;\vee\; \neg\, G[F@k].$$

If $\Phi_{P,\pi}$ is satisfiable, the SMT solver returns a model. Restricting this model to the variables at time 0 (the facts $F@0$) yields a counter-example. If unsatisfiable, then no counter-example exists and π is a valid plan.

3.1.3 SUPERB Algorithm

CPCES uses counter-examples to search for candidate plans, and we observe that the number of iterations required to find a valid plan heavily depends on the quality of the counter-examples found. For instance, in the running example, there are 25 locations (initial states). In the worst-case scenario, CPCES could potentially require 9 iterations—one for each column/row and one for last verifying—before finding a valid plan. However, in most cases, when CPCES selects counter-examples randomly, it will take more than 9 iterations. Therefore, the strategy for selecting counter-examples becomes crucial in improving the efficiency and keep robust of CPCES by minimizing the number of iterations. In (Zhang et al. 2020), we addressed this challenge by leveraging the concepts of *context* and *tag* from (Palacios and Geffner 2009) developing a method called SUPERB.

Definition 4. *A* subgoal *is a conjunct of the goal condition or of the precondition of some actions.*

Example 11. *In our running example, the goal condition is $x_3 \wedge y_3$. Therefore, there are two subgoals in the goal state: x_3 and y_3. In this example, the preconditions of the actions are all set to* ***True****, so no subgoal arises from the preconditions. However, if we consider the example from Figure 1.3, action* GO_E *and* GO_W *has precondition y_3. If the robot wants to move from (1,3) to (2,3) by executing action* GO_E*, the subgoal for this state transition is y_3. In a more complex situation, for example, if the precondition of an action is $f_1 \vee (f_2 \wedge f_3)$, there are two subgoals when applying this action: f_1 and $(f_2 \wedge f_3)$.*

Definition 5. *A fact f* depends on *another fact f' if there exists a conditional*

effect coneff $(a) = \langle c, \mathit{eff}^+, \mathit{eff}^- \rangle$ *such that* $f' \in c$ *and* $f \in \mathit{eff}^+ \cup \mathit{eff}^-$.

In this thesis, the notion of "depend on" is analogous the concept of "causal relevance" as defined by Bonet and Geffner (2014). In their framework, conformant planning problems are represented using a multi-valued encoding (Bäckström and Nebel 1995b), where each state variable can take one of several mutually exclusive values, and exactly one value must hold in any given state. In this setting, if the truth value of a fact f depends on another fact g, then g is considered causally relevant to f. This is analogous to our definition of "depend on", although our formalization is expressed in PDDL. Furthermore, Bonet and Geffner construct a "causal graph" by connecting causally relevant variables with directed edges. Conceptually, this plays the same role as our "dependency graph", which makes explicit the network of causal dependencies among facts.

Definition 6. *The* context $ctx(\phi)$ *associated with a subgoal* ϕ *comprises a set of facts and includes all the facts upon which they depend, extending transitively.*

Example 12. *Referring to the running example on computing contexts: In this scenario, as discussed in Example 11, there are two subgoals:* x_3 *and* y_3*. Consider the action* GO_E*: it has a conditional effect* $\langle \{x_2\}, \{x_3\}, \{x_2\} \rangle$*, indicating that upon moving East, the robot transitions from standing at* x_2 *to* x_3*. This effect informs us of the dependency of* $On(x_3, x_2)$*. Then we can further find the dependency of* $On(x_4, x_3)$*,* $On(x_5, x_4)$*,* $On(x_2, x_1)$*. These form a context* $ctx(x_3) = \{x_1, x_2, x_3, x_4, x_5\}$*. Another context identified in this problem is* $ctx(y_3) = \{y_1, y_2, y_3, y_4, y_5\}$.

Definition 7. *Given a context* c *(a simplified notation for* ctx*) and an initial state* s*, the tag* $t_c(s)$ *of* s *with respect to* c *is defined as the intersection of* s *and* c*, i.e.,* $t_c(s) = s \cap c$.

Example 13. *In Example 12, we identified two contexts* $ctx(x_3)$ *and* $ctx(y_3)$*, which are related to* x *and* y *respectively. Given an initial state* $s = \{x_2, y_2\}$*, the tag of* s *for* $ctx(x_3)$ *is* $\{x_2\}$ *and the tag of* s *for* $ctx(y_3)$ *is* $\{y_2\}$*. Thus, we say* s *contains two tags. Indeed, in this problem, every state contains two tags (one for each context).*

SUPERB strategically selects optimal counter-examples that contain as many new tags as possible. The rationale behind SUPERB is that each new tag provides

Algorithm 3 SUPERB version of CPCES.

```
1:  input: conformant planning problem P
2:  output: a conformant plan, or no plan
3:  B := ∅                                              ▷ empty sample set
4:  π := ϵ                                              ▷ empty plan
5:  loop
6:      q := generate_counter_example(P, π)
7:      if q ≠ ⊥ then
8:          return π
9:      loop
10:         q′ := improve_counter_example(B, q)
11:         if q′ = ⊥ then                              ▷ could not improve q any more
12:             break
13:         q := q′
14:     B := B ∪ {q}
15:     π := produce_candidate_plan(P, B)
16:     if π = ⊥ then
17:         return no plan
```

additional information about the context it belongs to, and the more new tags a counter-example contains, the more information CPCES can gather. SUPERB is particularly effective when there are at least two contexts in a problem, as with only one context, any counter-example would include only one new tag. This selection process is designed to offer more informative insights, enabling the identification of better candidate plans and ultimately reducing the number of iterations required by CPCES.

The SUPERB version of CPCES is presented in Algorithm 3. It is an improvement based on Algorithm 1, with the primary modifications occurring in Lines 9 to 13 of Algorithm 3. In each outer iteration (Line 5), SUPERB version of CPCES generates a random counter-example q (Line 6). The algorithm then enters an inner loop (Line 9) to improve this counter-example, refining it into q' (Line 10). The goal of this refinement is to ensure that the new counter-example q' (also called "new sample") contains more new tags compared to the original counter-example q. This process continues until no further new tags can be added (Line 11).

Here we explain how to improve counter-example (Line 10). Let B be the set of

previously found counter-examples (samples), $\mathcal{C}$ the set of contexts of P, let q be the counter-example generated at the current outer iteration (Line 6), let B' be the set of improved counter-examples (Line 10) collected at current iteration. We extend the same SMT encoding used for counter-examples by adding a improvement constraint.

SMT variables:

- $f@i$: for each fact $f \in F$ and time step $i \in \{0, \ldots, k\}$, representing whether f holds at step i.
- $\mathrm{Add}(f)@i$, $\mathrm{Del}(f)@i$: Boolean formulas indicating whether the ith action adds or deletes fact f.
- Ψ_{fail}: formula expressing that the plan fails (as in Section 3.1.2).
- $\Phi_{\mathsf{improve}}(B, q, \mathcal{C})$: new constraint enforcing novelty with respect to previously seen counter-examples.

Constraints: The base encoding is the same as for counter-examples:

$$\Phi_{P,\pi} = I[F@0] \ \wedge \ \Big(\bigwedge_{i=1}^{k} \Phi[F@(i-1), F@i] \Big) \ \wedge \ \Psi_{\mathsf{fail}}.$$

We now augment it with a constraint that requires at least one context to realize a new tag unseen so far.

For each context $c \in \mathcal{C}$, the tag of a state s is $t_c(s) = s \cap c$ (Definition 7). We collect all previously seen tags in that context as

$$\mathsf{Tags}_{\mathsf{prev}}(c) = \{\, t_c(ce) \mid ce \in B \cup \{q\} \cup B' \,\}.$$

To ensure novelty, we require at least one context c to realise a tag not in this set. For any $T \subseteq c$, let

$$\mathsf{TagEq}_c(T) = \Big(\bigwedge_{f \in T} f@0 \Big) \wedge \Big(\bigwedge_{f \in c \setminus T} \neg f@0 \Big),$$

and define the improvement constraint:

$$\Phi_{\mathsf{improve}}(B, q, \mathcal{C}) = \bigvee_{c \in \mathcal{C}} \Big(\bigwedge_{T \in \mathsf{Tags}_{\mathsf{prev}}(c)} \neg\, \mathsf{TagEq}_c(T) \Big).$$

The final SMT problem is

$$\Phi^{\mathsf{IMP}}_{P,\pi,B,q} = \Phi_{P,\pi} \ \wedge \ \Phi_{\mathsf{improve}}(B, q, \mathcal{C}).$$

We use an SMT solver to check satisfiability; any model restricted to $F@0$ yields an improved counter-example.

Table 3.1: Detailed process of how SUPERB version of CPCES solves the problem illustrated in Section 1.1.3.

Iteration	Sample	New Sample	Sample Set	Candidate Plan
0	-	-	$\emptyset$	ε
1	(2,2)	(2,2)	$\{(2,2)\}$	GO_E, GO_N
2	(2,5)	(5,5)	$\{(2,2),(5,5)\}$	GO_N $\times$ 3, GO_E $\times$ 3, GO_W $\times$ 2, GO_S $\times$ 2
3	(1,2)	(1,1)	$\{(2,2),(5,5),(1,1)\}$	GO_N $\times$ 4, GO_E $\times$ 4, GO_W $\times$ 2, GO_S $\times$ 2
4	-			

Example 14. *Let us use our running example to explain how* ***SUPERB*** *works.*

0. ***CPCES*** *starts with an empty sample set $B = \emptyset$ and an empty plan $\pi = \epsilon$.*

1. ***CPCES*** *finds a counter-example (2,2). This counter-example contains two tags: $\{x_2\}$ and $\{y_2\}$. Since these tags have never appeared before, this is considered a good counter-example.* ***CPCES*** *then finds a candidate plan: $\pi_1 =$* GO_E, GO_N.

2. *Based on π_1,* ***CPCES*** *finds a counter-example (2,5), which contains the tags $\{x_2\}$ and $\{y_5\}$. Since $\{x_2\}$ has appeared before, this is not a good counter-example. The counter-example is then improved to (5,5), where both tags $\{x_5\}$ and $\{y_5\}$ are new.* ***CPCES*** *then finds a new candidate plan: $\pi_2 =$* GO_N $*$ 3, GO_E $*$ 3, GO_W $*$ 2, GO_S $*$ 2.

3. *CPCES finds a counter-example (1,2). Since the tag $\{y_2\}$ has appeared before, this is not a good counter-example. The counter-example is improved to (1,1). CPCES then finds a new candidate plan:* $\pi_3 = \text{GO\_N} * 4, \text{GO\_E} * 4, \text{GO\_W} * 2, \text{GO\_S} * 2$.

4. *CPCES cannot find any new counter-examples. Therefore, π_3 is the solution, and the algorithm terminates.*

SUPERB has demonstrated remarkable performance in experimental results, particularly for problems with multiple contexts. It significantly reduces the number of iterations required to search for a plan, thereby decreasing the overall running time. However, for problems with a single context, SUPERB does not provide any advantage. The underlying reason for this phenomenon is that in problems with multiple contexts, the counter-example generated in each iteration can be continuously refined until it accumulates as many new tags as possible, eventually yielding an "optimal counter-example". In contrast, when a problem has only a single context, the counter-example in each iteration cannot be further improved.

In our regular use of CPCES, we observed that despite having the SUPERB version, there are still several areas that can be optimized to further enhance the search efficiency of CPCES. Thereby, we developed three additional improvements to CPCES. In the following sections, we will introduce each of these enhancements in detail.

3.2 Merging Certain-Facts in CPCES

A certain-fact is a fact whose value is always known during plan execution. Our first improvement to CPCES leverages the concept of certain-facts. As discussed in Section 3.1, during the candidate search part of CPCES, when translating a CP problem into a classical planning problem, all facts are cloned into a set of facts with interpretation numbers. However, since certain-facts retain the same value across all interpretations, their clones are redundant and do not provide additional information. Therefore, we can merge these facts, eliminating redundancy and reducing the total number of facts in the new PDDL files, which improves the efficiency of CPCES.

Definition 8. *A fact $f \in F$ is a* certain-fact *if for any plan π applicable in all initial states and for any pair (s, s') of initial states, the property $f \in s[\pi] \Leftrightarrow f \in s'[\pi]$ holds.*

Determining whether a fact qualifies as a certain-fact is a PSPACE-COMPLETE problem, as this determination hinges on whether two initially distinct states can be driven, under the same sequence of actions, to yield different truth values of the fact in question.[1] Consequently, we use an approximation method that relies on the conceptual framework of context from Palacios and Geffner (2009) and Bonet and Geffner (2014).

Definition 9. *The* dependency graph *is a directed graph whose vertices are the facts F and an edge from f to f' indicates that f depends on f'*

Algorithmically, computing dependency graph uses the same transitive-closure procedure as for computing a context. The only difference is that computing a dependency graph returns the full digraph, while computing a context returns just the set of nodes (facts) in that graph.

Whether a fact is a certain-fact or not can be easily identified from a dependency graph.

Example 15. *Figure 3.1 illustrates the process of identifying certain-facts from a dependency graph. The shaded nodes (a and e) represent facts that are initially unknown, while all other nodes (b, c, d, f, and g) are initially known. The directed edges indicate dependencies between the facts. There are four certain-facts: c, d, f, and g. The reason b is not a certain-fact is that it depends on e, which is not a certain-fact.*

Given a plan π, the belief is defined as the set of potential states the system might

[1] According to Definition 3.2, deciding that a fact f is *not* a certain-fact means there exist initial states s and s' and an applicable plan π such that $f \in s[\pi]$ and $f \notin s'[\pi]$. This can be checked by nondeterministically guessing such a pair (s, s') and invoking a classical planner to test whether a plan π exists that leads to a violating pair. Since each planning instance is solvable within polynomial space (Bylander 1994b), and guessing and simulating pairs of states also requires only polynomial space, deciding "f is a certain-fact" lies in PSPACE. For hardness, note that the problem subsumes classical planning: deciding whether there exists a plan that achieves a goal fact from an initial state is a special case of testing whether two executions (one with the goal fact **True** and one **False**) can diverge, hence the problem is PSPACE-hard. Therefore, determining certain-facts is PSPACE-COMPLETE.

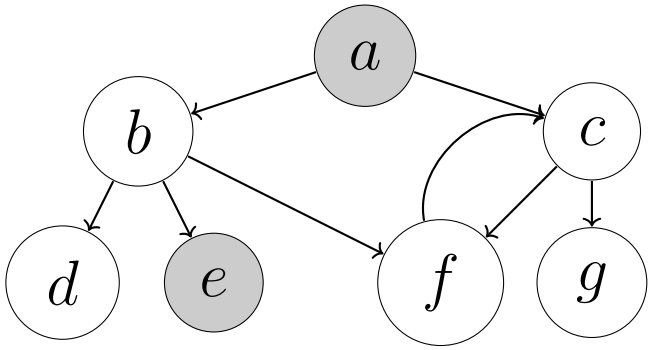


Figure 3.1: Illustration of certain-facts. Edges represent dependencies, and shaded nodes (a and e) denote facts that are initially unknown. Facts c, d, f, and g are identified as certain-facts.

be in, formally represented as $\mathcal{B} = \{s[\pi] \mid s \models I\}$. A fact is considered known in the initial belief if it is satisfied in all initial states: $\forall(s, s'), (s \models I) \wedge (s' \models I) \Rightarrow (f \in s \Leftrightarrow f \in s')$.

Lemma 1. *Let f be a fact such that all the facts it depends on (including itself) are known in the initial belief. Then f is a certain-fact.*

Proof of Lemma 1. Let F' be the set of facts that f depends on. We note that if f' is a fact from F', then all facts that f' depends on are in F'. We prove Lemma 1 by showing, through induction over the plans, that all facts in F' are certain-facts.

- Base Case ($\pi = \varepsilon$): This is the assumption of the lemma, that all facts in F' are known in the initial belief.
- Step Case: Assume all facts F' are known in the belief B_i reached by applying $\pi = a_1, \ldots, a_i$.

  Are all facts of F' known in the belief B_{i+1} reached by applying $a_1, \ldots, a_i, a_{i+1}$? By contradiction, assume that there is a fact $f' \in F'$ that is not known at step $i + 1$. Then, there are two states s_1 and s_2 from B_{i+1} where $f' \in s_1$ and $f' \notin s_2$.

  Let s'_1 and s'_2 be the two states from B_i such that $s_1 = s'_1[a_{i+1}]$ and $s_2 = s'_2[a_{i+1}]$. Since f' is known in B_i (induction hypothesis), f' is either in both s'_1 and s'_2, or in neither. Without loss of generality, assume $f' \in s'_1 \cap s'_2$. Since

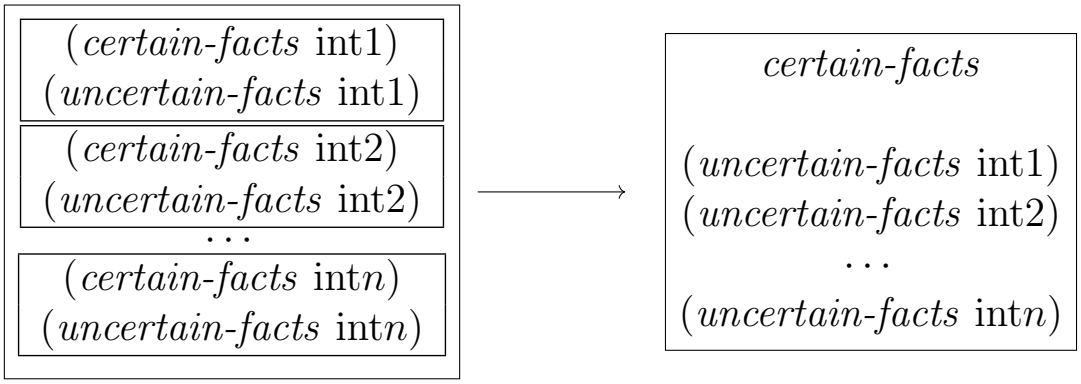


Figure 3.2: Certain-facts in the interpretation PDDL file can be merged and shared across all the interpretations.

f' is not in s_2, a conditional effect $\langle c, \mathit{eff}^+, \mathit{eff}^- \rangle$ of a_{i+1} must have triggered from state s'_2. However, we note that condition c only involves facts from F', which are all known in B_i. Therefore, the conditional effect also applies to s'_1, and f' should not be found in s_1, which contradicts the assumption.

This concludes the inductive step and, hence, the lemma. □

As discussed in Section 3.1, CPCES translates a CP problem into a classical problem by incorporating interpretation numbers into PDDL files, with each interpretation number representing a subproblem, as shown on the left side of Figure 3.2. We recognize that a certain-fact f is logically equivalent across all subproblems $\{(f\ int1), \cdots, (f\ intn)\}$, making it sufficient to maintain only one instance of these facts across all interpretations, as illustrated on the right side of Figure 3.2. This can be achieved by modifying the definitions of initial states, goals, and actions. As a result, the PDDL file becomes more compact, leading to improved planner performance. For instance, heuristic-search planners are expected to achieve greater accuracy in their heuristic estimates.

Example 16. *Let us consider Figure 1.3 in Section 1.1.5 (not our running example), which offers a more complex example. If we set the robot to stand on the top row y_3 of the grid (where x is unknown), then all the facts in $\{y_1, y_2, y_3\}$ are certain-facts, and there is no need to clone them in each iteration. However, if we set the robot to stand on the leftmost column x_1 of the grid (where y is unknown), x must still be cloned in each iteration because* GO_E *and* GO_W *has preconditions.*

For instance, applying GO_E *from* $s = \{x_1, y_3\}$ *leads to* $\{x_2, y_3\}$ *while applying* GO_E *from* $s' = \{x_1, y_2\}$ *leads to* $\{x_1, y_2\}$. *In other words, although* s *and* s' *have the same* x *value,* $s[\text{GO\_E}]$ *and* $s[\text{GO\_E}]$ *do not.*

Our method for merging certain-facts removes per-interpretation clones for facts that remain known across all interpretations during plan execution — i.e., we keep a single shared instance of such facts in the compiled task. In contrast, the T0 planner (Palacios and Geffner 2009) uses tagged knowledge literals (facts) KL/t and avoids tag-conditioned copies *ex ante*:

- T0 does not create KL/t when the tag closure t^* contains no literal relevant to L, replacing every occurrence of KL/t by KL since $KL \subset KL/t$;
- T0 skips support/cancellation rules conditioned on non-empty tags when they cannot contribute to any merge that covers a precondition/goal literal;
- When there is no initial uncertainty about L given t, the T0 translation groups the support and cancellation rules as $KC/t \rightarrow KL/t \wedge \neg K \neg L/t$, hence KL/t (or $K \neg L/t$) holds as an invariant in the compiled task.

Thus, both mechanisms reduce the compiled task by eliminating redundant conditioned copies.

3.3 Warm-Starting CPCES

As mentioned in Section 3.1, the number of iterations required to find a valid plan heavily depends on the quality of the counter-examples generated. To address this, we developed SUPERB to ensure that CPCES finds an optimal counter-example in each iteration, thereby decreasing the number of iterations needed. Although SUPERB is highly powerful, it still cannot minimize the number of iterations required to find a valid plan. In our running example, SUPERB takes up to five iterations (in each iteration, updating both x and y in the counter-example) to find a plan. We observed that some initial states are more critical than others. For instance, if CPCES selects $\{x_1, y_1\}$ and $\{x_5, y_5\}$ as the first two counter-examples, a valid plan can be found immediately, reducing the number of iterations to two. In fact, to solve this problem within two iterations, we can compute a set of counter-

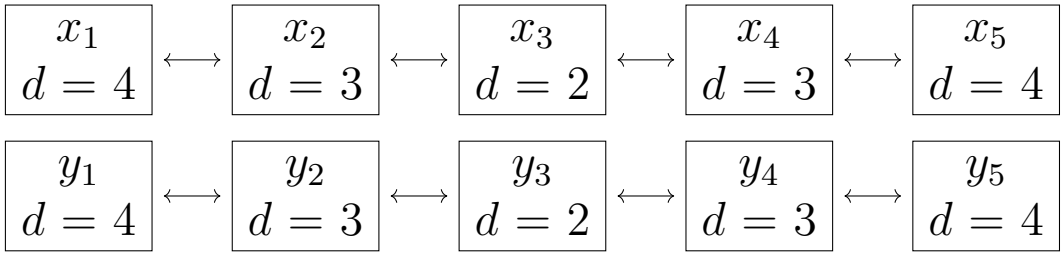


Figure 3.3: Two dependency graphs with the scores of the facts in the running example from Section 1.1.3. Here, x and y are two facts refer to the abscissa and the ordinate of the robot, and d is the score.

examples that cover all tags in the CP. However, this set of counter-examples may still be unnecessarily large. Therefore, our goal is to identify a minimal set that is still good. In our running example, four initial states can be considered best: combinations of x_1, x_5, and y_1, y_5 (e. g., $\{x_1, y_1\}$, $\{x_1, y_5\}$, $\{x_5, y_1\}$, $\{x_5, y_5\}$). Why are these better than other initial states? Upon closer observation, you will notice they are all located at the corners of the grid. When the distance between two locations (counter-examples) is maximized, it forces the agent to consider the possible locations between them, leading to the quickest speed of finding a valid plan. Inspired by this insight, we developed the warm-starting CPCES.

Warm-starting CPCES is built on the concept of the dependency graph, as defined in Definition 9. Recall that if a fact f depends on another fact f', there must exists an action $a = \langle c, \mathit{eff}^+, \mathit{eff}^- \rangle$ such that $f \in \mathit{eff}^+ \cup \mathit{eff}^-$ and $f \in c$. This implies that the greater the distance between two facts in a dependency graph, the more actions are required to transition from one fact to the other. Therefore, if these two facts are included in counter-examples, more initial conditions will be covered by the candidate plan.

Definition 10. *The* distance *from fact f to fact f' is the minimal number of steps from f to f' in the dependency graph. If there is no such path, the distance is assumed to be zero. The score of a fact f is the largest distance from this fact to any other fact.*

Example 17. *We can construct two dependency graphs, as shown in Figure 3.3, for our running example. In the first dependency graph, x_1 directly depends on x_2 but indirectly depends on other facts. The largest distance from x_1 to any other is*

4, which is from x_1 to x_5. Thus, we assign a score of 5 to x_1. The scores for other facts are computed in a similar way.

Definition 11. *An* uncertain-fact *is called an important fact if it maximises the score within some dependency graphs. If all uncertain-facts of an initial state are important, this state is important.*

Warm-starting CPCES begins by initializing the sample set with important samples, and we require the size of sample set be smallest: each important fact appears only once. The warm-starting process is described in Algorithm 4. First, we calculate the score for each fact to identify the important facts (Line 4). Then, using an SMT solver, we iteratively search for initial states that contain previously unconsidered important facts (Line 7). This process continues until no new important states are found. Since no two initial states share the same important fact, this procedure eventually terminates.

To compute important samples, we extend the same SMT encoding used for counter-examples by adding a covering constraint.

SMT variables:

- $f@i$ for each fact $f \in F$ and $i \in \{0, \ldots, k\}$, representing state variables.
- $\mathrm{Add}(f)@i$ and $\mathrm{Del}(f)@i$, capturing addition and deletion of facts.
- Ψ_{fail}, indicating plan failure.

Constraints: The base encoding is

$$\Phi_{P,\pi} = I[F@0] \ \wedge \ \Big(\bigwedge_{i=1}^{k} \Phi[F@(i-1), F@i] \Big) \ \wedge \ \Psi_{\mathsf{fail}}.$$

Let M be the set of important facts, $\mathcal{C}$ the set of contexts of P, and $M' \subseteq M$ the subset already used. We require that each context with unused important facts contributes at least one such fact at time 0:

$$\Phi_{\mathsf{cover}}(M, \mathcal{C}, M') = \bigwedge_{\substack{ctx \in \mathcal{C} \\ M \cap ctx \neq \emptyset}} \bigvee_{f \in (M \cap ctx) \setminus M'} f@0.$$

Algorithm 4 Generating important samples.

```
1: input: conformant planning problem P
2: output: a set of important samples
3: S := compute_scores(P)
4: F := pickup_important_facts(P, S)
5: B := ∅
6: loop
7:     q := generate_important_sample(B, F)
8:     if q ≠ ⊥ then
9:         return B
10:    B := B ∪ {q}
```

Final SMT problem:

$$\Phi^{\mathsf{WS}}_{P,\pi} = \Phi_{P,\pi} \ \wedge \ \Phi_{\mathsf{cover}}(M, \mathcal{C}, M').$$

Any satisfying assignment, restricted to the variables at time 0 ($F@0$), yields an important sample.

3.4 Integrating Fast Downward Into CPCES

FF system (Domshlak and Hoffmann 2007) is used in CPCES to find candidate plans. In fact, all classical planners can be used in CPCES, and this includes FD system (Helmert 2006). FD is regarded as a versatile planner due to its wide array of heuristics and planning techniques. However, preliminary tests reveal that FD is significantly slower than FF as a classical sub-planner in our setting. This performance disparity is shown in Table 2, where the NMF (FF) and FD (FD) columns are compared. Except for one case, the FF implementation consistently surpasses FD, often by several orders of magnitude. We tried many different configurations of FD, including the LAMA-2011 configuration (Richter et al. 2011), but the results were very similar.

We identified that the poor performance of FD in CPCES is due to the slow translation of PDDL files into SAS+ files during each iteration of CPCES. Specifically, there are two issues. First, as the number of iterations increases, the size of the

Table 3.2: Number of variables in the SAS+ problem with and without forall.

Domain	Instance	With	Without
BLOCKWORLD	p02	115	42
BOMB	p20-5	625	117
COINS	p15	903	432
DISPOSE	p4-2	641	33
LOOK-GRAB	p4-2-2	442	14
ONEDISPOSE	p3-2	500	76
RAOSKEYS	p2	39	18

PDDL (forall construction is included) files grows, making it increasingly challenging for FD to perform the translation into SAS+ files. This is illustrated in Table 3.2. In this experiment, we replace forall with an explicit enumeration of the interpretations in the PDDL file. We find the version without forall requires much fewer variables in SAS+. The current representation limits the FD search engine's ability to fully exploit its capabilities. Second, each iteration of CPCES involves translating from PDDL to SAS+, which incurs a significant cost. To mitigate these issues, we decided to incrementally construct the SAS+ file. To clarify this process, we first introduce the concept of SAS+.

3.4.1 SAS+

Instead of using facts as basic elements, like PDDL, SAS+ uses multi-valued encoding to represent a planning problem (Bäckström and Nebel 1995b; Helmert 2009). In this encoding, the basic element is the state variable. Each state variable v has a domain D_v. A state is defined as a complete assignment of the state variables. An atomic effect $(v = \nu)$, which assigns the value ν to the variable v, corresponds to a positive effect $(v = \nu)$ and a list of negative effects $(v = \nu')$ for all other values $\nu' \in D_v \setminus \nu$.

Translating a PDDL representation to a SAS+ representation generally involves identifying subsets of pairwise mutually exclusive facts, denoted as F' (Helmert 2006). For each such subset, a variable v is created with the domain $D_v = F' \cup \{\bot\}$. In a SAS+ state, variable v is assigned to $f \in F'$ if and only if the fact f appears in

the equivalent PDDL state, represented as $s \cap F' = \{f\}$. The symbol $\perp$ represents a situation in which none of the facts from F' are present in the state, denoted as $s \cap F' = \emptyset$.

One advantage of multi-valued encoding is that it offers a more compact state representation (Bäckström and Nebel 1995b). Additionally, it enables the development of richer heuristic functions (Geffner 2007; Richter et al. 2008; Richter and Westphal 2010).

3.4.2 Incremental SAS+

Definition 12. *Two planning problems P_1 and P_2, where $P_i = \langle F_i, N_i, A_i, I_i, G_i \rangle$ for $i \in \{1, 2\}$, are* independent *if their sets of action names are identical and their sets of facts are disjoint: $(N_1 = N_2) \wedge (F_1 \cap F_2 = \emptyset)$.*

We also introduce the definition of *merge*, which combines two problems that share the same set of action names.

Definition 13. *Let P_1 and P_2 be two planning problems where $P_i = \langle F_i, N_i, A_i, I_i, G_i \rangle$ for $i \in \{1, 2\}$. The* merge *of P_1 and P_2, denoted $P_1 \boxplus P_2$, is a problem $P = \langle F, N, A, I, G \rangle$ defined by:*

- $F = F_1 \cup F_2$; $\quad I = I_1 \wedge I_2$; $\quad G = G_1 \wedge G_2$;
- $N = N_1 = N_2$; *and*
- A *is such that* $A(a) = \langle pre_1(a) \wedge pre_2(a), coneff_1(a) \cup coneff_2(a) \rangle$ *for all* $a \in N$.

Definition 13 naturally extends to the SAS+ notation, and we therefore use the operator $\boxplus$ for SAS+ problems as well. We denote this as $P \boxplus P_\Delta$, where P_Δ represents an *increment* to the *original problem* P. An increment of a problem refines an existing planning problem by defining new facts relevant to the task and specifying the interactions of actions with these new facts. Additionally, the increment has the characteristic of not directly affecting existing facts.

The concepts of merge and increment offer significant advantages by facilitating a straightforward translation to SAS+ when a translation for the original

problem has already been established. This can be encapsulated in the following lemma:

Lemma 2. *Let P be a PDDL planning problem, and let P_Δ be an increment to P. Let S and S_Δ be SAS+ representations of P and P_Δ respectively such that*

- $\Pi(S) = \Pi(P)$*;*
- $\Pi(S_\Delta) = \Pi(P_\Delta)$*; and*
- *the sets of variables of S and S_Δ are disjoint.*

Then $S \boxplus S_\Delta$ is a SAS+ representation of $P \boxplus P_\Delta$ such that $\Pi(S \boxplus S_\Delta) = \Pi(P \boxplus P_\Delta)$.

Proof of Lemma 2. The proof is based on the principle that when two independent planning problems, whether represented in PDDL or SAS+, are merged, the valid solutions for the combined problem consist of the intersection of the solutions from the original problems. Hence,

$$\begin{aligned}\Pi(S \boxplus S_\Delta) &= \Pi(S) \cap \Pi(S_\Delta) \\ &= \Pi(P) \cap \Pi(P_\Delta) \\ &= \Pi(P \boxplus P_\Delta).\end{aligned}$$

Based on Lemma 2, we have developed a method for efficiently generating SAS+ representations of the classical planning problems formulated by CPCES. In each iteration, the combined classical planning problem $P_{1\ldots i}$ is constructed by merging $P_{1\ldots i-1}$ with P_i, incorporating all classical planning problems from P_1 to P_i which correspond to the i interpretations or counter-examples generated up to that point in the process. The SAS+ representation of $P_{1\ldots i}$ is achieved by merging the SAS+ representation from the previous CPCES iteration of $P_{1\ldots i-1}$ with the SAS+ representation of P_i. This process is depicted in Figure 3.4, where the vertical operation that converts $P_{1\ldots i}$ to $S_{1\ldots i}$ involves decomposing and then merging each segment, depicted on the right and left side of the figure, respectively.

Our approach is detailed in Algorithm 5, which implements produce_candidate_plan method (Line 15) from CPCES (Algorithm 1). In this algorithm, a dictionary is used to track the SAS+ translation for each state considered in the belief. When the algorithm finds a new state (Line 9), it will

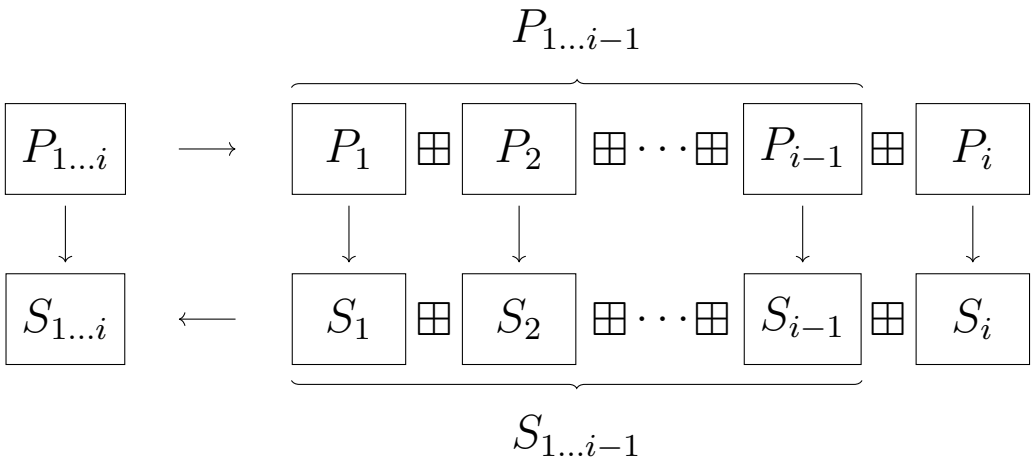


Figure 3.4: Graphical representation of the decomposition proposed in this paper: the classical planning problem $P_{1...i}$ that CPCES(FF) needs solved at the ith iteration can be decomposed into i planning problems $P_{j,j\in[1...i]}$. Each of these problems can be translated into its own SAS+ representation, S_j, and these representations merged into a SAS+ representation $S_{1...i}$ equivalent to $P_{1...i}$.

compute the SAS+, i.e., $S(q)$, for this state (Line 10), and then merge each state's individual SAS+ (Line 11).

There are several benefits to this approach. First, it is very fast to generate SAS+. In CPCES, if we use FD to compute candidate plan, SAS+ is generated from $P_{1...i}$ which takes a long time. However, in each iteration, this approach only needs to translate a small PDDL P_i into a small SAS+ S_i, which is much faster than translating a large PDDL $P_{1...i}$ into a large SAS+ $S_{1...i}$. Second, this approach avoids using the forall construct, which presents a significant challenge for FD when computing candidate plans. Third, since each P_i problem is relatively small, we can allocate additional resources to optimize the SAS+ encoding. This is something that would be riskier to attempt for the complete problem $P_{1...i}$.

Inspired by the concept of merging facts in PDDL, merging variables in SAS+ proves to be equally effective. By leveraging the method of identifying certain-facts in PDDL, we can identify variables that contain facts that are initially known and have no impact on uncertain-facts. In SAS+, certain-facts within variables maintain a fixed Boolean value from the initial state to the goal state. When all

Algorithm 5 Incremental generation of candidate plan.

1: **method**: `produce_candidate_plan`(P, B)
2: **input**: conformant planning problem $P = \langle F, N, A, I, G\rangle$
3: **input**: belief B
4: **output**: a candidate plan, or **no plan**
5: **static**: a map $S : 2^F \rightarrow$ SAS_FILES
6: **if** $B = \emptyset$ **then**
7: **return** ε
8: **for all** $q \in B$ **do**
9: **if** $S(q)$ is undefined **then**
10: $S(q) :=$ `generate_sas_plus`$(\langle F, N, A, q, G\rangle)$
11: $SAS := \boxplus_{q\in B} S(q)$
12: **return** `FD`(SAS)

the facts within a variable are certain-facts, we refer to that variable as a certain-variable. By applying the same merging process used for certain-facts in PDDL to certain-variables in SAS+, `FD` can reduce the search space and significantly improve search efficiency when generating a plan.

A similar idea can be found in the translation scheme $K_{T,M}$ of Palacios and Geffner (2009). In their framework, uncertainty in the initial situation is handled by introducing tags $t \in T$ and merge actions over these tags, while preconditions and goals are expressed using literals of the form KL/t. In particular, the goal conditions are represented by introducing copies of the relevant facts for each possible initial assumption, thereby ensuring that a plan achieves the goal under all initial situations. This design makes incremental combination possible: since each assumption carries its own copy of the goal fact, new assumptions can be incorporated and their constraints merged without invalidating the existing encoding. Our incremental SAS+ formulation follows the same underlying principle — by representing goals with assumption — specific copies of the facts, we enable an iterative construction where additional counter-examples can be accommodated incrementally rather than requiring a complete re-encoding from scratch.

3.5 Experimental Results

The benchmarks used in our experiments are the same seven sets of benchmarks (DISPOSE, ONEDISPOSE, BLOCKWORLD, LOOK-GRAB, RAOSKEYS, BOMB, and COINS) previously employed in published CPCES studies (Zhang et al. 2020). These benchmarks are originally taken from the benchmarks of the International Planning Competition 2008. This consistency ensures that our experimental results are more objective and comparable to prior work. These experiments were performed on an Ubuntu virtual machine equipped with 8GB of RAM and a single processor on an Apple M1 chip MacBook Pro. The time limit for each test was set to 1800 seconds. Given the non-deterministic nature of the CPCES process, each instance was run 9 times, with the median result being reported. The CPCES used in this chapter is the SUPERB version of CPCES.

In this subsection, we have assigned abbreviated names to each improved conformant planning algorithm. NMF is the CPCES (FF used); MF is the variant with merging certain-facts (including constants); WS is warm-starting with merging certain facts (including constants); FD is CPCES (FD used); ICC is incremental CPCES (FD used). The overall performance of runtimes of each method is presented on Figure 3.5

3.5.1 Merging Certain-Facts

We first empirically evaluated the performance gains attributable to the merging of certain-facts improvement to CPCES we described in Section 3.2. We are primarily interested in contrasting planning time, with experimental results given in Table 3.3.

For most problems MF is much faster than NMF. For example, in problem LOOK-GRAB p8-3-1 MF is twice as fast as NMF. In problem DISPOSE p12-1, NMF fails to find a plan within 30 minutes while MF quickly finds a solution in 478 seconds. Even in problems without certain-facts the improvement may still have an impact, because constants are also exploited with this improvement. For instance, in problem BOMB p100-100, which has no certain-facts, solving it with MF is almost twice as fast as with NMF due to the occurrence of 99% of problem facts

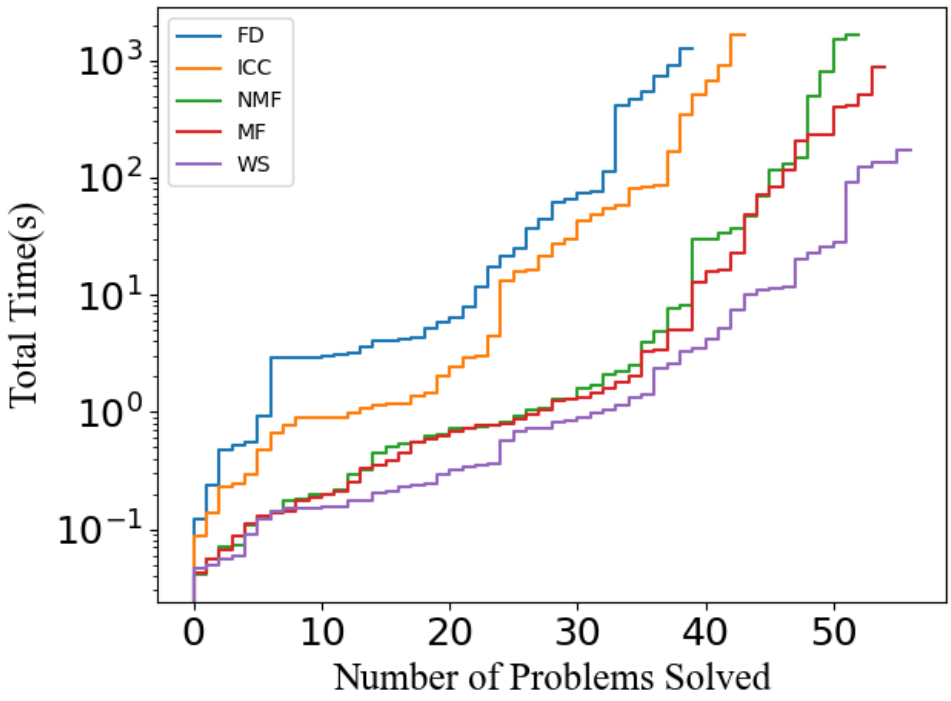


Figure 3.5: Number of problems solved by each algorithm.

as constant.

In our implementation, constants are subsumed under the definition of certain-facts (Definition 3.2). Merging such constants has the effect of simplifying facts that never change, thereby avoiding repeated encodings across iterations. If constants were not considered during the merging process, the algorithm would remain sound and complete, since the omission does not alter the semantics of the problem. However, the encoding would become less compact, as many redundant predicates would remain in the PDDL description, which in turn increases the runtime of the underlying classical planner. This effect can be clearly observed in the `BOMB` p100-100, where no explicit certain-facts are present but the exploitation of constants still results in nearly a twofold speedup.

Experimentally we found no clear relationship between the speedup afforded by the improvement in MF and the certain-facts ratio. For example, in `DISPOSE` p8-1, p8-2, and p8-3, the certain-facts ratios are 0.5, 0.33, and 0.25, respectively,

but the MF runtimes are 44%, 70%, and 45%, respectively, of the NMF runtimes. What complicates the relationship is that as the number of iterations to compute a conformant plan increases the classical PDDL files become lengthier. The runtime performance of the underlying classical planner degrades, typically superlinearly with the length of the input problem description.

Table 3.3: Empirical performance of total time (seconds) of improved conformant planning algorithms described in the paper along with the CPCES baseline results.

DM	Instance	NMF	MF	WS	FD	ICC
BLOCKWORLD	01	0.04	0.04	0.05	0.12	0.09
BLOCKWORLD	02	0.18	0.18	0.15	0.48	0.29
BLOCKWORLD	03	1.62	1.86	1.39	-	-
BLOCKWORLD	04	109.59	231.65	157.38	-	-
BOMB	20-1	0.46	0.46	0.16	5.06	0.77
BOMB	20-5	0.57	0.54	0.18	7.76	-
BOMB	20-10	0.76	0.69	0.21	11.39	1.16
BOMB	20-20	1.11	1.08	0.36	24.07	2.00
BOMB	100-1	33.24	22.94	1.05	-	21.19
BOMB	100-5	69.23	48.62	3.45	-	341.67
BOMB	100-10	116.08	72.57	5.15	-	898.75
BOMB	100-60	789.79	430.02	27.92	-	-
BOMB	100-100	1636.60	899.29	91.96	-	-
COINS	10	0.14	0.14	0.15	0.46	0.24
COINS	12	1.51	1.50	0.90	6.18	1.49
COINS	15	0.87	0.96	0.86	5.76	0.87
COINS	16	0.72	0.77	0.71	2.95	0.96
COINS	17	0.65	0.69	0.73	2.81	0.96
COINS	18	0.63	0.63	0.69	2.82	0.90
COINS	19	0.81	0.83	0.86	2.85	0.88
COINS	20	0.75	0.72	0.94	2.90	0.95
COINS	21	-	-	-	-	-

Continued on next page

DM	Instance	NMF	MF	WS	FD	ICC
DISPOSE	4-1	0.54	0.40	0.12	2.91	0.67
DISPOSE	4-2	1.06	0.88	0.14	3.56	1.16
DISPOSE	4-3	1.35	1.39	0.17	4.04	1.46
DISPOSE	8-1	37.45	16.31	1.14	711.99	15.74
DISPOSE	8-2	130.22	91.52	2.37	870.44	57.67
DISPOSE	8-3	431.88	195.67	7.54	1243.76	86.56
DISPOSE	12-1	-	478.08	11.06	-	514.97
DISPOSE	12-2	-	-	124.35	-	1686.45
DISPOSE	12-3	-	-	-	-	-
DISPOSE	16-1	-	-	156.57	-	-
DISPOSE	16-2	-	-	-	-	-
RAOSKEYS	02	0.07	0.07	0.06	0.24	0.14
RAOSKEYS	03	0.71	0.57	0.57	64.42	13.88
LOOK-GRAB	4-1-1	0.29	0.25	0.21	3.10	2.38
LOOK-GRAB	4-1-2	0.07	0.08	0.15	36.86	26.90
LOOK-GRAB	4-1-3	0.05	0.06	0.15	21.33	29.93
LOOK-GRAB	4-2-1	0.21	0.25	0.23	2.86	2.88
LOOK-GRAB	4-2-2	0.12	0.14	0.24	74.08	48.42
LOOK-GRAB	4-2-3	0.11	0.11	0.25	43.10	54.18
LOOK-GRAB	4-3-1	0.31	0.36	0.31	6.26	15.98
LOOK-GRAB	4-3-2	0.20	0.20	0.34	111.54	53.30
LOOK-GRAB	4-3-3	0.18	0.19	0.36	65.19	83.62
LOOK-GRAB	8-1-1	7.68	5.03	4.04	355.60	162.93
LOOK-GRAB	8-1-2	2.31	1.57	3.29	-	-
LOOK-GRAB	8-1-3	1.45	1.25	2.57	-	-
LOOK-GRAB	8-2-1	34.65	14.14	11.70	503.33	-
LOOK-GRAB	8-2-2	4.85	3.29	12.46	-	-
LOOK-GRAB	8-2-3	2.45	2.07	7.81	-	-
LOOK-GRAB	8-3-1	33.82	14.53	25.77	450.82	-

Continued on next page

DM	Instance	NMF	MF	WS	FD	ICC
LOOK-GRAB	8-3-2	5.54	5.04	22.35	-	-
LOOK-GRAB	8-3-3	3.98	3.41	-	-	-
ONEDISPOSE	2-2	0.19	0.13	0.05	0.51	0.27
ONEDISPOSE	2-3	0.47	0.29	0.06	0.94	0.44
ONEDISPOSE	3-2	2.50	1.31	0.09	4.16	1.22
ONEDISPOSE	3-3	50.99	224.79	0.36	16.01	3.02
ONEDISPOSE	4-2	-	72.31	1.26	73.25	4.17
ONEDISPOSE	4-3	-	-	-	-	441.88
ONEDISPOSE	5-2	-	454.77	178.50	-	42.19
ONEDISPOSE	5-3	-	-	-	-	-
ONEDISPOSE	6-2	-	-	-	-	1367.08
ONEDISPOSE	6-3	-	-	-	-	-

3.5.2 Warm-Starting CPCES

In Section 3.3 we described how CPCES can further be improved with warm starting. The results of warm-starting CPCES are presented in Table 3.3 and 3.4, under the WS column. That system is implemented based on MF, and thus has FF as the classical base planner and implements merging of certain facts. Below we focus on the runtime gains over MF we obtain with warm starting.

Under the most ideal circumstances, if enough critical samples are identified at beginning (Line 4 in Algorithm 4) CPCES can find a valid solution in just two iterations: the first iteration finds a candidate plan, and the second verifies that no more counter-examples exist. Indeed, from Table 3.3 and 3.4 we observe that most problems are solved by WS within two iterations. This indicates that our warm-starting CPCES variant efficiently identifies sufficient important samples at the outset, allowing it to find a valid plan immediately and significantly reducing the plan search time. For example, in problem DISPOSE p8-3, WS finds a plan in 7.54s over 2 iterations, while MF takes 195.67s across 65 iterations. For some problems, although they are not solved within just two iterations, the number of iterations and runtime is significantly lower compared to MF. For instance, in problem BOMB p100-100, WS takes 91.96s to find a plan within 4 iterations. In contrast, MF spends 899.29s computing 100 iterations. In cases where more than 2 iterations are required by WS, we thus still can observe a speedup. Warm-starting does not however work for all problems. For example, in the COINS domain there is no noticeable reduction in iterations. In the LOOK-GRAB domain, although the number of iterations is reduced to 2, the plan search time is actually worse than with MF, with the warm-start impacting the relative difficulty of the underlying classical problem.

3.5.3 Integrating FD Into CPCES

Finally we consider the empirical performance attributes of CPCES using the FD base planner as described in Section 3.4. The results of integrating FD into CPCES are presented in Table 3.3 and 3.4. The search engine used by our FD integration is an eager best-first search algorithm with a best-first open list and the FF heuristic.

We compared using FF in CPCES (column NMF), referred to as CPCES(FF), with using FD in CPCES, referred to as CPCES(FD) (column FD). We further contrast the performance of those two base systems with our FD based incremental version of CPCES(FD), referred to as I-CPCES(FD).

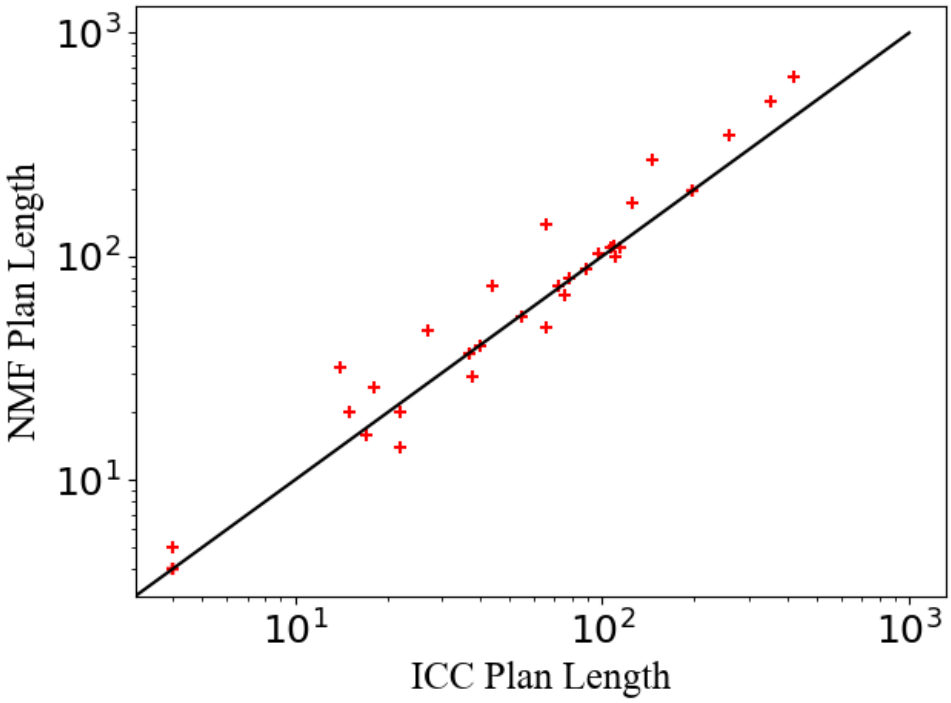


Figure 3.6: Comparison of plan length between ICC and NMF.

CPCES(FD) is not generally competitive with CPCES(FF) with the preprocessing PDDL to a multi-valued state variable representation to be a key bottleneck. I-CPCES(FD) shows great improvement compared to CPCES(FD). For instance, I-CPCES(FD) requires 86.56s to solve DISPOSE p8-3, not only significantly faster than CPCES(FD) but also five times faster than CPCES(FF).

Although incrementality using the FD based planner can yield impressive performance, overall we find that the FF base planner generally exhibits better runtime performance. This is particularly noticeable in the challenging domains of BLOCKWORLD, BOMB, LOOK-GRAB, and RAOSKEYS. For instance, problem BOMB p100-10 takes I-CPCES(FD) 898.75s, while CPCES(FF) takes only 116.08s. To explain this, we can compare the plan lengths generated by CPCES(FF) and I-CPCES(FD)

(Figure 3.6) and found that FD consistently attempts to find shorter plans than CPCES(FF) (e.g., 419 steps vs. 469 for DISPOSE p8-3 or 66 steps vs. 148 for ONEDISPOSE p3-3). In contrast to the FF-based system, the FD-based planner is obtaining shorter plans at a higher search cost which negatively impacts overall runtime performance. We note that the overall performance of I-CPCES(FD) is still worse than CPCES(FF) (Figure 3.5), especially in complex domains such as BLOCKWORLD, BOMB, LOOK-GRAB, and RAOSKEYS.

Table 3.4: Empirical results on the number of iterations and plan lengths for improved conformant planning algorithms, in comparison to the CPCES baseline. The last column reports the ratio of certain-facts (RF) for each problem.

DM	Instance	Iterations					Plan Length		RF
		NMF	MF	WS	FD	ICC	NMF	ICC	(%)
BLOCKWORLD	01	3	3	3	3	3	5	4	0
BLOCKWORLD	02	8	8	6	8	8	18	15	0
BLOCKWORLD	03	29	29	23	-	-	70	-	0
BLOCKWORLD	04	94	86	82	-	-	152	-	0
BOMB	20-1	21	21	3	21	21	40	40	0
BOMB	20-5	20	20	3	21	-	37	-	0
BOMB	20-10	20	20	3	20	20	37	37	0
BOMB	20-20	20	20	4	20	20	37	37	0
BOMB	100-1	100	100	2	-	100	197	197	0
BOMB	100-5	100	100	3	-	100	197	197	0
BOMB	100-10	100	100	3	-	100	197	197	0
BOMB	100-60	100	100	3	-	-	197	-	0
BOMB	100-100	100	100	4	-	-	197	-	0
COINS	10	6	6	6	6	6	29	38	0
COINS	12	22	23	16	22	22	67	76	0
COINS	15	14	16	15	21	13	80	78	0
COINS	16	10	10	9	11	10	93	113	0
COINS	17	9	9	9	11	11	96	98	0

Continued on next page

DM	Instance	Iterations					Plan Length		RF
		NMF	MF	WS	FD	ICC	NMF	ICC	(%)
COINS	18	9	9	9	11	10	103	101	0
COINS	19	10	11	10	11	10	110	107	0
COINS	20	10	9	10	10	10	100	109	0
COINS	21	-	-	-	-	-	-	-	-
DISPOSE	4-1	17	17	2	17	17	61	55	-
DISPOSE	4-2	17	17	2	17	17	74	72	33
DISPOSE	4-3	17	17	2	17	17	84	89	25
DISPOSE	8-1	65	65	2	65	65	399	259	50
DISPOSE	8-2	65	65	2	65	65	482	354	33
DISPOSE	8-3	65	65	2	65	65	469	419	25
DISPOSE	12-1	-	145	2	-	145	-	697	50
DISPOSE	12-2	-	-	2	-	145	-	912	33
DISPOSE	12-3	-	-	-	-	-	-	-	-
DISPOSE	16-1	-	-	2	-	-	-	-	50
DISPOSE	16-2	-	-	-	-	-	-	-	-
RAOSKEYS	02	5	5	4	5	5	16	17	14
RAOSKEYS	03	21	18	19	37	37	48	66	11
LOOK-GRAB	4-1-1	9	9	2	7	8	32	14	-
LOOK-GRAB	4-1-2	3	3	2	4	4	4	4	-
LOOK-GRAB	4-1-3	2	2	2	2	2	4	4	-
LOOK-GRAB	4-2-1	6	7	2	5	6	16	22	-
LOOK-GRAB	4-2-2	3	4	2	4	4	4	4	-
LOOK-GRAB	4-2-3	2	2	2	2	2	4	4	-
LOOK-GRAB	4-3-1	6	7	2	6	6	22	22	-
LOOK-GRAB	4-3-2	3	3	2	4	3	4	4	-
LOOK-GRAB	4-3-3	2	2	2	2	2	4	4	20
LOOK-GRAB	8-1-1	18	18	2	22	24	172	80	33
LOOK-GRAB	8-1-2	13	13	2	-	-	80	-	33

Continued on next page

DM	Instance	Iterations					Plan Length		RF
		NMF	MF	WS	FD	ICC	NMF	ICC	(%)
LOOK-GRAB	8-1-3	10	11	2	-	-	52	-	33
LOOK-GRAB	8-2-1	20	22	2	19	-	98	-	25
LOOK-GRAB	8-2-2	11	9	2	-	-	70	-	25
LOOK-GRAB	8-2-3	9	9	2	-	-	50	-	-
LOOK-GRAB	8-3-1	18	16	2	15	-	134	-	20
LOOK-GRAB	8-3-2	10	11	2	-	-	60	-	20
LOOK-GRAB	8-3-3	8	9	-	-	-	40	-	-
ONEDISPOSE	2-2	10	8	2	9	9	26	18	-
ONEDISPOSE	2-3	17	12	2	14	13	43	27	20
ONEDISPOSE	3-2	29	26	2	29	26	74	44	25
ONEDISPOSE	3-3	45	44	5	57	51	148	66	20
ONEDISPOSE	4-2	-	61	11	80	51	-	80	25
ONEDISPOSE	4-3	-	-	-	-	112	-	115	20
ONEDISPOSE	5-2	-	141	81	-	173	-	146	25
ONEDISPOSE	5-3	-	-	-	-	-	-	-	-
ONEDISPOSE	6-2	-	-	-	-	247	-	228	-
ONEDISPOSE	6-3	-	-	-	-	-	-	-	-

3.6 Conclusions

In this chapter, we present three methods for improving the efficiency of the CPCES algorithm. These enhancements aim to accelerate candidate plan generation and integrate FD into CPCES, thereby improving overall planning performance.

The first method involves merging certain-facts to reduce the number of predicates used in the classical abstractions. Experimental results demonstrate that this approach significantly enhances the performance of CPCES. The second method introduces a warm-starting mechanism, built upon merging certain-facts, to CPCES. Before entering the main loop, the algorithm identifies a set of important initial states. By exploring candidate plans from these selected states, CPCES is able to discover valid plans with fewer iterations. Empirical evidence shows that, for most benchmark problems, warm-starting enables CPCES to find a valid plan within just two iterations (one to generate a candidate plan and one to validate it). For other problems, relatively few iterations are required, resulting in a substantial reduction in overall solving time. The third method integrates the FD planner into the CPCES framework. To address the performance bottleneck caused by the inefficient generation of SAS+ representations in FD, we propose an incremental merging strategy for small SAS+ instances. This approach allows the system to construct the final SAS+ task representation more efficiently, avoiding the overhead associated with generating large grounded representations all at once. Experimental results indicate that our integrated method outperforms the baseline CPCES approach that naively invokes FD, achieving faster solution times across a range of conformant planning problems.

Chapter 4

Counter-Example PCP Planners

We have demonstrated that the counter-example based approach can be used to solve CP. In this chapter, we present how this approach can be applied to solve PCP. We named this approach as `p-CPCES`. `p-CPCES` is a PCP planner inspired by the work of Grastien and Scala (2020) and Huang (2006). We will use the definition of PCP from Definition 3 and use the example from Chapter 1 Section 1.1.3 as the running example throughout this chapter.

4.1 Challenges of Using the `CPCES` Approach

Simply copying `CPCES` to solve PCP is not complete. The reason is that, if the procedure aims to iteratively find plans with a higher probability of success, each call to the classical planner must result in a solvable classical problem. However, if states from which the goal is unreachable are included in the starting states, the resulting classical problem becomes unsolvable. Imagine some locations in our running example as traps — once the agent moves there, it will never exit. If one of these initial states is selected as a counter-example, no candidate plan can be found. In this case, `CPCES` must restart the iteration and hope that none of these problematic initial states are selected as counter-examples. Only when no such states are chosen can a candidate plan be found. Some PCP problems may be even more extreme — if specific combinations of initial states prevent the agent from reaching the goal condition, `CPCES` will need to restart the entire process, beginning with the first iteration. In our running example, there are $2^9 - 1$ combinations of initial states, meaning the counter-example based approach might

need to go through $2^9 - 1$ iterations before finding a valid plan. To avoid exploring all combinations, in this section, we introduce the concept of a "counter-tag" to reduce the number of iterations.

Aside from the issue mentioned above, calculating the satisfaction probability of a candidate plan presents another challenge. We use *Deterministic Decomposable Negation Normal Form* (d-DNNF) as a data structure that makes this practical, as the time complexity of computing probabilities using d-DNNF is polynomial (Darwiche and Marquis 2002b,a; Huang 2006).

4.2 Overview of p-CPCES

Since a tag is the intersection of a context and an initial state (Definition 7), a single tag can correspond to multiple different initial states. In other words, a tag can represent a set of initial states. For example, in our running example, we can identify two contexts in this problem: $\{x_1, x_2, x_3\}$, $\{y_1, y_2, y_3\}$, and six tags: $\{x_1\}$, $\{x_2\}$, $\{x_3\}$, $\{y_1\}$, $\{y_2\}$, $\{y_3\}$. Tag $\{x_1\}$ represents three initial states, which are $\{x_1, y_1\}$, $\{x_1, y_2\}$, and $\{x_1, y_3\}$. While the number of combinations of initial states is $2^9 - 1$, the number of combinations of tags is only $2^6 - 1$. This means that focusing on tags rather than individual initial states can significantly reduce the number of iterations. Therefore, our approach computes "*counter-tags*" instead of counter-examples. Since all tags in our running example are singletons, from now, we simplify the notation by omitting the brackets. For instance, we use tag x_1 rather than $\{x_1\}$ to represent a tag over context $\{x_1, x_2, x_3\}$.

A counter-tag for a plan π is a tag such that all initial states it represents are counter-examples to π. For instance, if $\pi = \text{GO\_E}$, then x_3 is a counter-tag because all counter-examples corresponding to x_3 (i.e., $\{x_3, y_1\}$, $\{x_3, y_2\}$ and $\{x_3, y_3\}$) are counter-examples to π. A *counter-tag set* (CTS) is a set of counter-tags that covers the initial belief with a probability of at least $(1 - \tau)$. A candidate plan for a CTS should satisfy one of counter-tags in the CTS, rather than needing to satisfy all counter-tags in the CTS. This mechanism ensures that the candidate plan addresses any initial state that is unreachable.

Let us use our running example to demonstrate how our approach works, as illus-

Table 4.1: Example execution of p-CPCES for the running example with $\tau = 0.75$.

# it	Counter-tags	CTS	Probability	Candidate Plan
0	-	-	-	ε
1	x_1, x_3, y_1, y_3	$\{x_1, x_3\}$	.3	GO_W
2	x_1, x_2, y_1, y_2, y_3	$\{x_2\}$	.7	GO_W GO_E
3	x_3, y_1, y_2, y_3	$\{y_2\}$	.7	GO_N GO_E GO_W GO_S
4	x_1, y_1	$\{x_1, y_1\}$	.28	GO_N GO_W GO_E GO_S
5	x_3, y_1	None	N/A	-

trated in Table 4.1.

0. At the beginning, we set the candidate plan as an empty plan $\pi_0 = \varepsilon$.

1. Four tags are counter-tags to π_0, which are x_1, x_3, y_1, and y_3. It is unnecessary to consider all counter-tags in this iteration; instead, we only consider x_1 and x_3, as the combined probability of these two counter-tags is 0.3, which exceeds 1-0.75. Thus, we add x_1 and x_3 into CTS_1. A candidate plan for CTS_1 is GO_W, which satisfies x_3 in CTS_1, and we update the candidate plan to $\pi_1 =$ GO_W.

2. Five tags are now counter-tags to π_1. As in the previous step, we select x_2 as a counter-tag for this iteration and add it to CTS_2. From this iteration onward, the candidate plan must satisfy at least one tag in the current iteration's CTS, as well as at least one tag in the CTS of each previous iteration. We then update the candidate plan to $\pi_2 =$ GO_W, GO_E, which is consistent with both x_1 (from CTS_1) and x_2 (from CTS_2). Since CTS_2 contains only one tag, future plans will ensure that x_2 is no longer a counter-tag.

3. p-CPCES continues running, finding increasingly sophisticated candidate plans.

4. In this round, the CTS is $\text{CTS}_4 = \{x_1, y_1\}$. The probability of the initial states represented by x_1 is 0.2, and for y_1, it is 0.1. However, the combined probability of both is not 0.2 + 0.1 = 0.3, because x_1 and y_1 share the overlapping initial state $\{x_1, y_1\}$, which has a probability of $0.2 \times 0.1 = 0.02$. Thus, the actual probability of CTS_4 is 0.3 - 0.02 = 0.28. The candidate plan is updated to $\pi_4 =$ GO_N, GO_WGO_E, GO_S.

5. In the final iteration, even though two counter-tags x_3 and y_1 remain, their combined probability is 0.19, which is less than $1 - \tau = 0.25$. Since there are not enough counter-tags to π_4, π_4 must be a valid plan for this problem. `p-CPCES` terminates and returns π_4.

Algorithm 6 `p-CPCES`

```
1: Input: a PCP problem 𝒫
2: Output: a plan π for 𝒫, or UNSAT
3: B := ∅                                          ▷ a set of counter-tags
4: π := ε                                          ▷ candidate plan
5: loop
6:     CTS := compute_CTS(𝒫, π)                    ▷ Section 4.3
7:     if CTS = ⊥ then
8:         return π
9:     B := B ∪ {CTS}
10:    π := compute_candidate_plan(𝒫, B)           ▷ Section 4.5
11:    if π = ⊥ then
12:        return no plan
```

`p-CPCES` is outlined in Algorithm 6. The set B collects the counter-tag sets that have been computed so far, and the candidate plan π begins with an empty plan ε. `p-CPCES` generates a new counter-tag set; if none exists (all counter-tags are identified but the total probability mass of initial states corresponding to these counter-tags is less than $1-\tau$), the current candidate plan is a solution. Otherwise, the set is added to B. Following this, `p-CPCES` computes a candidate plan that aligns with at least one tag from each counter-tag set in B. If no such plan can be found, the planning problem has no solution. Otherwise, a new iteration begins.

4.3 Computing Counter-Tags

The concept of counter-tags is derived from the idea of a projected planning problem. A projection limits the planning problem to a specific context and sets the initial state to a tag. We define the projection in a bottom-up manner, starting from simple elements and building up to more complex structures.

Recall Definition 7, where a tag is defined as the intersection of an initial state s and a context c, i.e., $t_c(s) = s \cap c$. Below, we represent a tag together with its context as a pair $ct = \langle c, t \rangle$. The set of tags of $\mathcal{P}$ is denoted by $Tags(\mathcal{P})$.

Definition 14. *Let $ct = \langle c, t \rangle$ be a tag.*

- *The* projection of a conjunction $\varphi = \bigwedge_{j \in \{1,\ldots,k\}} \varphi_j$ over ct *is*

$$Proj(\varphi, ct) = \bigwedge_{j \in \{1,\ldots,k\}.vars(\varphi_j) \subseteq c} \varphi_j$$

- *The* projection of a set of conditional effects *coneff* over ct *is the subset of conditional effects*

$$\{\langle con, c \cap \mathit{eff}^+, c \cap \mathit{eff}^- \rangle \mid \langle con, \mathit{eff}^+, \mathit{eff}^- \rangle \in \mathit{coneff} \wedge vars(con) \subseteq c\}$$

- *We overload the Proj notation and for the* projection of action a over ct *we have*

$$Proj(a, ct) = \langle name(a), Proj(pre(a), ct), Proj(\mathit{coneff}(a), ct) \rangle$$

- *The* projection of $\mathcal{P} = \langle F, A, I, G, \tau \rangle$ over ct *is the classical planning problem* $Proj(\mathcal{P}, ct) = \mathcal{P}_{ct}$ *defined as follows:*
  - $F_{ct} = ct \cup cons(P)$*, a set of facts relevant in the context ct, including all facts appearing in the context ct, and all* constant facts *($cons(P)$), i.e., facts that do not appear in any context and thus are assumed to be* ***True*** *or relevant in all contexts.*
  - $A_{ct} = \{Proj(a, ct) \mid a \in A\}$*, a set of projected actions. Each action $Proj(a, ct)$ in A_{ct} has the same name as its counterpart a in A, but its conditional effects are projected based on the context ct. If an action has no conditional effects, it is still included in A_{ct} as a projected action.*
  - $I_{ct} = \{t\}$*, the initial state, represented as a set of facts given by the tag t. That is, only the facts included in the tag t are considered* ***True*** *initially.*

– $G_{ct} = Proj(G, ct)$, *the projected goal condition. It is a conjunction of facts from the original goal* G*, projected onto the context* ct*.*

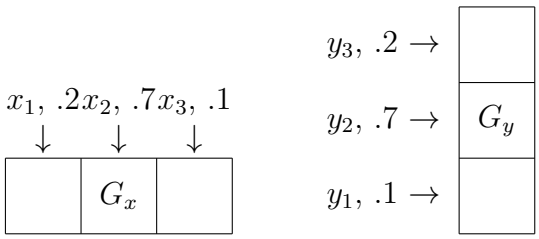


Figure 4.1: Two projected problems derived by projecting the PCP problem from Section 1.1.5 onto different contexts. The left projected problem pertains to x with sub-goal G_x, and the right one pertains to y with sub-goal G_y.

Example 18. *In our running example, since all facts belong to a context there is no constant fact in this problem. Because there are two contexts, we can derive two projected problems. The first projected problem (left side of Figure 4.1) is related to* x *with the sub-goal being* x_2*, while the second projected problem (right side of Figure 4.1) is related to* y*, with the sub-goal being* y_2*.*

As you can see, after the problem is projected it is unnecessary to consider each initial state individually. Instead, we can focus on tags since, in the original problem, each initial state is a combination of several tags, whereas in the projected problem, the initial state corresponds to a single tag. However, the projection of the problem raises a new concern: the preconditions of each action may not be valid in each projected problem. For instance, in the running example, actions GO_E and GO_W have the precondition "when located at y_3". But in the first projected problem (left in Figure 4.1), where there is only one row in the grid, it is unclear when GO_E and GO_W can be applied. To address this issue, we introduce the concept of "trivialization" of the planning problem (we also call them *trivial problems* or *sub-problems*), where the preconditions of actions are moved into the conditional effects of the corresponding action.

Definition 15. *The* trivialisation *of PCP problem* $\mathcal{P}$ *is the planning problem* $Triv(\mathcal{P}) = \mathcal{P}' = \langle F', A', I', G' \rangle$ *defined by*

- $F' = F \cup \{\zeta\}$ *where* $\zeta \notin F$ *is a new fact;*

- $A' = \{Triv(a) \mid a \in A\}$, *where* $Triv(a) = \langle name(a), \textbf{\textit{True}}, coneff(a)\rangle$ *if* $pre(a) = \textbf{\textit{True}}$, *and* $Triv(a) = \langle name(a), \textbf{\textit{True}}, coneff(a) \cup \langle \neg pre(a), \emptyset, \{\zeta\}\rangle\rangle$ *otherwise;*
- $I' = \{(i \cup \{\zeta\} \mapsto v) \mid (i \mapsto v) \in I\} \cup \{(i \mapsto 0) \mid i \subseteq F\}$;
- $G' = G \wedge \zeta$.

In other words, the trivialization did two things: (i) it introduces a new fact, ζ which value is set as **True** initially, into the problem, which acts as an additional sub-goal that must be **True** after executing all actions of the plan; (ii) it removes the precondition, *pre*, from the actions and adds a conditional effect to the same action. This effect states that if *pre* is not satisfied, ζ is assigned the value **False**, which causes the goal to remain unsatisfiable. This mechanism preserves the function of the precondition while avoiding its direct use. We will introduce how to use the trivialization combined with problem projection in Section 4.5.

Example 19. *In our running example, we introduce a new fact, (valid), which is in the initial condition. After trivialization, the action* GO_E *will no longer have any preconditions but will include one additional conditional effect: "when the agent is not located at* y_3*, the fact (valid) becomes* ***False****". The goal after trivialization is* $(x_2) \wedge (y_2) \wedge (valid)$.

Lemma 3. *Let* $\mathcal{P}$ *be a planning problem. The following holds:* $\Pi(\mathcal{P}) = \Pi(Triv(\mathcal{P}))$.

Proof of Lemma 3. We write $\mathcal{P}' = Triv(\mathcal{P})$. For this proof, we concentrate on a single initial state i for $\mathcal{P}$ and the corresponding initial state $i' = i \cup \{\zeta\}$ for $\mathcal{P}'$; the result then generalizes to the initial state probability distribution.

It should be clear that, for any sequence of actions $\pi = a_1, \ldots, a_k$, the state $i'[\pi]$ equals $i'[\pi] \setminus \{\zeta\}$ (but note that this does not imply $\zeta \in i[\pi]$) since the conditional effects of these actions are similar except for ζ.

Consider a sequence of actions $\pi = a_1, \ldots, a_n$ that is a valid plan to $\mathcal{P}$ but not a solution to $\mathcal{P}'$. Let us prove that there is a conflict. Since all actions in $\mathcal{P}'$ are trivial, and since both the states $i[\pi]$ and $i'[\pi]$, and goals G and G' differ only on ζ, we have $\zeta \notin i'[\pi]$. Furthermore, since $\zeta \in i'$, there must be an action, a_j, that

makes ζ as a negative effect. This means that the condition φ that triggers this negative effect is not satisfied; however, φ is the negation of the precondition of a_j in $\mathcal{P}$. Therefore, since the states $i[a_1, \ldots, a_{k-1}]$ and $i'[a_1, \ldots, a_{k-1}]$ are identical except for ζ, the action a_j is not applicable in $\mathcal{P}$ and π is not valid from i.

Consider a sequence of actions $\pi = a_1, \ldots, a_n$ that is a valid plan to $\mathcal{P}'$. Let us prove that π is a valid plan to $\mathcal{P}$. Because π is valid to $\mathcal{P}'$, ζ is **True** in the final state. This implies that none of the conditions that have ζ has a negative effect trigger (there is an action that would make ζ positive again), i.e., the negations of these conditions are always **True**. These negations are exactly the preconditions of the actions in $\mathcal{P}$, and we know that the states $i[a_1, \ldots, a_k]$ and $i'[a_1, \ldots, a_k]$ are identical except for ζ. Therefore, π is valid in $\mathcal{P}$. □

Definition 16. *A* counter-tag *to π is a tag $ct \in Tags(\mathcal{P})$ such that $\pi \notin \Pi(Proj(\mathcal{P}, ct))$. The set of all counter-tags to π is $CTags(\pi)$.*

In other words, a counter-tag for a plan π is a tag where all the initial states it represents are counter-examples to π. A counter-tag can be identified by executing the plan on a tag and checking if, at any step, the precondition of an action or the goal becomes invalid. For a tag $ct = \langle c, t \rangle$, we write $[\![ct]\!]$ to denote the set of initial states i that satisfy this tag. Furthermore, given a set C of tags, we write $[\![C]\!]$ to represent the set of initial states that satisfy at least one of these tags: $[\![C]\!] = \bigcup_{ct \in C} [\![ct]\!]$.

Theorem 1. *The set of states in which candidate plan π is invalid is $[\![CTags(\pi)]\!]$.*

Proof of Theorem 1. **Part A:** We prove if an initial state $i \in [\![CTags(\pi)]\!]$, then π is invalid for that initial state i. Based on $[\![CTags(\pi)]\!] = \bigcup_{ct \in CTags(\pi)} [\![ct]\!]$, we can have $i \in \bigcup_{ct \in CTags(\pi)} [\![ct]\!]$. Since $\forall ct \in CTags(\pi)$, $\pi \notin \Pi(Proj(\mathcal{P}, ct))$, we get that π is invalid for i.

Part B: We prove if π is invalid for an initial state i, then $i \in [\![CTags(\pi)]\!]$. When i is an initial state that is unsatisfied with π, there must be a subgoal φ that is not satisfied during the execution of the plan. Let c be a context of this subgoal. By definition of a tag, the satisfaction of the subgoal at any time of the execution is defined entirely by the tag that was true in the initial state. Therefore, any other

state that has the same tag will also fail the validity test. Hence, this tag is a counter-tag, i.e., the state i belongs to $[\![CTags(\pi)]\!]$. □

Verifying whether a tag is a counter-tag in `p-CPCES` can be achieved by using an SMT solver, which checks whether, by executing a plan, the tag satisfies all action preconditions and meets the goal condition. However, verifying whether the probability of a set of counter-tags exceeds the threshold $1-\tau$ is more challenging. Inspired from the approach by Huang (2006), we used d-DNNF to compute this probability.

The computation of counter-tags in PCP mirrors that of counter-examples in CP (Section 3.1.2). We first remove probabilistic aspects: each distribution in the initial condition is treated as a oneof group, and the goal threshold τ is ignored. The resulting PCP problem has the same Boolean structure as a CP problem, so we reuse the same encoding.

SMT encoding:

$$\Phi_{P,\pi} = I[F@0] \ \wedge \ \Big(\bigwedge_{i=1}^{k} \Phi[F@(i-1), F@i] \Big) \ \wedge \ \Psi_{\mathsf{fail}}.$$

A tag is a set of facts $t \subseteq F$, represented at time 0 as

$$t[F@0] = \bigwedge_{f \in t} f@0.$$

For each tag t, we check

$$\Phi^{t}_{P,\pi} = \Phi_{P,\pi} \wedge t[F@0].$$

If $\Phi^{t}_{P,\pi}$ is satisfiable, t is a counter-tag.

Given a set of tags B, we test each $t \in B$; when satisfiable, t is added to the counter-tag set CTS. We maintain the cumulative probability mass of CTS, and stop once it exceeds $1-\tau$.

All checks are performed by an SMT solver. Any satisfying model, restricted to $F@0$, confirms that the plan π fails for tag t.

4.4 Compute the Probability of a Set of Initial States

In this section, we demonstrate how a set of states $[\![C]\!]$ can be compiled into a Deterministic Decomposable Negation Normal Form (d-DNNF) (Darwiche and Marquis 2002b; Darwiche 2001; Darwiche and Marquis 2002a; Huang 2006). Building on this compilation, we will show how to compute the probability of these states $[\![C]\!]$.

If we use the traditional mathematical approach, the probability of each initial state is obtained by multiplying the probabilities of its facts (for example, in our running example, the probability of (x_1, y_1) is $0.1 \times 0.2 = 0.02$). However, computing the probability of a set of states this way requires enumerating all possible initial states, which is exponential in complexity. Specifically, with n oneof distributions and m facts in each distribution, the number of possible initial states is n^m, leading to an exponential number of multiplications. This becomes impractical for complex PCP problems. The situation is even worse in `p-CPCES`, since at each iteration, whenever a new counter-tag is found, the probability of the accumulated counter-tags needs to be recomputed. To overcome this limitation, we rely on a d-DNNF compilation, which provides a compact factorization of the set of states. Within this representation, adjusting the weights of the chance variables ensures that the probabilities are assigned correctly while enabling polynomial-time evaluation (Darwiche 2001).

A propositional formula is in d-DNNF if it (i) exclusively uses conjunction, disjunction, and negation, with negation appearing only next to variables, and (ii) adheres to the principles of decomposability and determinism. Decomposability means that the conjuncts in any conjunction must not share any variables, while determinism requires that the disjuncts in any disjunction must be pairwise logically inconsistent. The advantage of compiling a set of initial states into a d-DNNF and then compute the probability of that it provides an explicit and compact factorization of the state set, which is provably small. This factorization allows for efficient evaluation, enabling us to compute the required probability in polynomial time relative to the size of the formula.

Let us first see how to compute the probability of a single initial state using a traditional mathematical method. Given an initial state i (recall from Definition 3 that all facts $f \in i$ are considered **True**, so $i = f_1 \cup \cdots \cup f_n$), some subsets $F'_j \subseteq i$ are associated with *oneof probability distribution* M_j. The probability of the initial state i is the product of the probabilities of each subset in the corresponding oneof probability distribution: $P = (M_1 \cap F'_1) \times \cdots \times (M_n \cap F'_n)$.

Example 20. *In our running example, there are two oneof probability distributions, corresponding to x and y respectively. To compute the probability of the initial state $\{x_1, y_3\}$, we use the product rule by multiplying* 0.2 *and* 0.1*, resulting in 0.02, which is the probability of $\{x_1, y_3\}$.*

$$M_x = \begin{cases} \{x_1\} & \mapsto & 0.2 \\ \{x_2\} & \mapsto & 0.7 \\ \{x_3\} & \mapsto & 0.1 \end{cases} \quad M_y = \begin{cases} \{y_1\} & \mapsto & 0.2 \\ \{y_2\} & \mapsto & 0.7 \\ \{y_3\} & \mapsto & 0.1 \end{cases}$$

The example above computes the probability of a single initial state. When computing a large set of initial states, however, this method becomes very slow. Let's explore how to use d-DNNF to perform this computation more efficiently.

Probabilities associated with facts in oneof probability distributions are captured by a specific set of propositional variables, referred to as chance variables. Consider a oneof probability distribution M_j over F_j, and let $\{i_1, \ldots, i_k\}$ represent the domain of M_j (i.e., $M_j(i_p)$ is defined for all $p \in \{1, \ldots, k\}$). Given a set of k chance variables $X_j = \{\chi_{j,1}, \ldots, \chi_{j,k}\}$, for each index p, we define a formula associating $\chi_{j,p}$ with i_p as follows:

$$\varphi_{j,p} = \left(\chi_{j,p} \wedge \bigwedge_{q<p} \neg\chi_{j,q} \right) \rightarrow \left(\bigwedge_{f \in i_p} f \wedge \bigwedge_{f \in F_j \setminus i_p} \neg f \right).$$

Finally, we express φ_j as:

$$\varphi_j = \bigwedge_{p \in \{1,\ldots,k\}} \varphi_{j,p}$$

In other words, for any Boolean assignment α of $F \cup X_j$ that satisfies φ_j, if $\chi_{j,\ell}$

is the chance variable with the smallest index ℓ-index such that $\alpha[\chi_{j,\ell}]$ is **True**, then the state i represented by α satisfies $i \cap F_j = i_j$. Finally, we must assign the correct probabilities to the chance variables, which are computed as follows:

$$\chi_p \mapsto M_j(i_p) / \left(1 - \sum_{q<p} M_j(i_q)\right).$$

Given a formula φ that represents a set of states (such as the set $[\![C]\!]$), the probability that the initial state satisfies φ can be calculated as:

$$count(\exists F.\ \varphi \wedge \bigwedge_{j\in\{1,\ldots,m\}} \varphi_j).$$

Example 21. *Let us now look how to compute and construct chance variables in our running example. For the oneof probability distribution M_x, there should be three chance variables: χ_{x1}, χ_{x2}, and χ_{x3}. The logic relationship between each fact in M_x and the corresponding chance variables is as follows:*

$$\begin{cases} x_1 \mapsto \chi_{x1} \\ x_2 \mapsto \chi_{x2} \wedge \neg\chi_{x1} \\ x_3 \mapsto \chi_{x3} \wedge \neg\chi_{x1} \wedge \neg\chi_{x1} \end{cases}$$

Accordingly, the value of each chance variable can be computed as:

$$\begin{cases} \chi_{x,1}{=}0.2 \\ \chi_{x,2}{=}0.7/(1-0.2) = 0.875 \\ \chi_{x,3}{=}1 \end{cases}$$

We apply the same procedure to compute the chance variables for M_y:

$$\begin{cases} y_1 \mapsto \chi_{y1} \\ y_2 \mapsto \chi_{y2} \wedge \neg\chi_{y1} \\ y_3 \mapsto \chi_{y3} \wedge \neg\chi_{y1} \wedge \neg\chi_{y1} \end{cases} \quad \begin{cases} \chi_{y,1}{=}0.2 \\ \chi_{y,2}{=}0.7/(1-0.2) = 0.875 \\ \chi_{y,3}{=}1 \end{cases}$$

Once the Boolean value of each chance variable is determined, it corresponds to a unique initial state. The probability of this initial state is the product of the values of the chance variables that are assigned as ***True****.*

We propose a method that compiles a PCP problem into a logic formula φ, in which the probability of a single initial state can be obtained simply by assigning values to the chance variables. To compute the probability of a large set of initial states, it is unnecessary to calculate each one individually and sum them. Instead, given a set of counter-tags C, the initial states associated with the chance variables represented by C can be expressed as:

$$\varphi \wedge \bigvee_{c \in C} c$$

This formula is then input into a d-DNNF compiler `DSHARP` (Muise et al. 2012). Using Formula *count*, the probability of initial states represented by the counter-tags can be easily computed.

Algorithm 7 Computing the Probability of Initial States.

$$val(N) = \begin{cases} Pr(p), & \text{if } N \text{ is a leaf node } p \text{ where } p \text{ is a chance variable;} \\ 1 - Pr(p), & \text{if } N \text{ is a leaf node } \neg p \text{ where } p \text{ is a chance variable;} \\ \prod_i val(N_i), & \text{if } N = \bigwedge_i N_i; \\ \sum_i val(N_i), & \text{if } N = \bigvee_i N_i. \end{cases}$$

Algorithm 17 provides the function for evaluating a node in the diagram, and thereby illustrates how to compute the probability of initial states satisfying a d-DNNF – i.e., by evaluating the root node of a diagram. After compiling the formula $\varphi \wedge \bigvee_{c \in C} c$ into a d-DNNF, the result is a tree-like data structure where each leaf node corresponds to either a chance variable p or a negated chance variable $\neg p$. The internal nodes represent either conjunctions or disjunctions. To compute the probability, we begin by assigning values to the leaf nodes: if a node corresponds to a chance variable p, the node is assigned the probability of p: $Pr(p)$; if it corresponds to a negated chance variable $\neg p$, the node is assigned $1 - Pr(p)$. The computation then proceeds recursively in a bottom-up manner. For a conjunctive

node, its value is computed as the product of the values of its child nodes; for a disjunctive node, its value is the sum of the values of its child nodes. This recursive process continues until the value of the root node is computed. The value of the root node represents the probability of initial states satisfying the given formula.

4.5 Computing Candidate Plans

In section 4.2, we have used an example to show how to compute a candidate plan. In this section, we will present the details of this progress (Line 10 in Algorithm 6). The basic idea is that we first decompose the problem into *sub-problems* (we also call them *trivial problems*) based on the contexts. This step has been introduced in Section 4.3. In each iteration, depending on the counter-tags selected, we choose certain trivial problems and merge them into a single problem (we call it *composite problem*). The plan that is valid for this composite problem problem becomes the candidate plan for that iteration. In the following paragraphs, we will explain how to merge trivial problems to compute the candidate plans.

The following operations assume that the planning problems have disjoint sets of facts. If this is not the case, we assume the facts are renamed to ensure this condition is met.

Definition 17. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two classical planning problems with $F_1 \cap F_2 = \emptyset$ and $\{name(a) \mid a \in A_1\} = \{name(a) \mid a \in A_2\}$ (The conditions, preconditions and conditional effects, of actions in $\mathcal{P}_1$ and $\mathcal{P}_2$ are identical except for the interpretation number associated with predicates and propositional symbols. $\mathcal{P}_1$ uses a unique interpretation number, as does $\mathcal{P}_2$). The* conjunction of $\mathcal{P}_1$ and $\mathcal{P}_2$ *is the classical planning problem $\mathcal{P} = \mathcal{P}_1 \otimes \mathcal{P}_2$ where*

- $F = F_1 \cup F_2$, $i = i_1 \cup i_2$, $G = G_1 \wedge G_2$, *and*
- $A = \{a_1 \otimes a_2 \mid a_1 \in A_1 \wedge a_2 \in A_2 \wedge name(a_1) = name(a_2)\}$ *where* $a = a_1 \otimes a_2$ *is defined by*
  - $name(a) = name(a_1)$,
  - $pre(a) = pre(a_1) \wedge pre(a_2)$, *and*

– $\mathit{coneff}(a) = \mathit{coneff}(a_1) \cup \mathit{coneff}(a_2)$.

$\otimes$ is called "conjunction" because of the following result:

Lemma 4. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two classical planning problems. The following holds:* $\Pi(\mathcal{P}_1 \otimes \mathcal{P}_2) = \Pi(\mathcal{P}_1) \cap \Pi(\mathcal{P}_2)$

Proof of Lemma 4. **Part A:** We first need to prove $\Pi(\mathcal{P}_1) \cap \Pi(\mathcal{P}_2) \subseteq \Pi(\mathcal{P}_1 \otimes \mathcal{P}_2)$. We do that by picking a plan π that is valid for $\mathcal{P}_1$ and $\mathcal{P}_2$ and show that it is valid for $\mathcal{P}_1 \otimes \mathcal{P}_2$.

Let $\pi = a_1, \ldots, a_n$. Assume π is valid for $\mathcal{P}_1$: $s_{1,0} \xrightarrow{a_1} s_{1,1} \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_{1,n}$ and also valid in $\mathcal{P}_2$: $s_{2,0} \xrightarrow{a_1} s_{2,1} \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_{2,n}$.

For all $j \in \{0, \ldots, k\}$, let $s_j = s_{1,j} \cup s_{2,j}$. We shall prove $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$. Note that the corresponding actions a_i in $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_1 \otimes \mathcal{P}_2$ share the same action name by definition. Moreover, their preconditions and conditional effects involve predicates that have the same names but are associated with different interpretation indices.

We need to prove the four following points:

1. Prove s_0 is the initial state i of $\mathcal{P}_1 \otimes \mathcal{P}_2$. Based on Definition 17, we can get $s_0 = s_{1,0} \cup s_{2,0} = i$.
2. Prove $\forall k$ a_k is applicable in s_{k-1}. The precondition of a_k is the conjunction of the preconditions from $\mathcal{P}_1$ and $\mathcal{P}_2$ which are independent (rely on different facts). Since they are satisfied in $s_{1,k-1}$ and $s_{2,k-1}$, and these states rely on different facts, the conjunction is satisfied in the union state.
3. Prove $\forall k$ $a_k[s_{k-1}] = s_k$. Again, the conditions of the conditional effects are independent for the two sub-problems $\mathcal{P}_1$ and $\mathcal{P}_2$, and their effects are independent. The conditional effects of the action therefore act independently.
4. Prove s_n is a goal state. Similar to Point 2 on preconditions, the goal condition is the conjunction of the goal conditions in either sub-problem, and the union of the two independent sets $s_{1,n} \cup s_{2,n}$ satisfies the conjunction.

Part B: We then need to prove $\Pi(\mathcal{P}_1 \otimes \mathcal{P}_2) \subseteq \Pi(\mathcal{P}_1) \cap \Pi(\mathcal{P}_2)$. Without loss of generality, we proceed to prove noting the proof for $\Pi(\mathcal{P}_1 \otimes \mathcal{P}_2) \subseteq \Pi(\mathcal{P}_1)$ follows analogously.

Let $\pi = [a_1, \ldots, a_n]$ be a plan for $\mathcal{P}_1 \otimes \mathcal{P}_2$, i.e., such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$. For all $j \in \{0, \ldots, k\}$, let $s_{1,j} = s_j \cap F_1$. We need to prove the four properties that guarantee $s_{1,0} \xrightarrow{a_1} s_{1,1} \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_{1,n}$. Again, the corresponding actions a_i in $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_1 \otimes \mathcal{P}_2$ share the same action name by definition.

1. Prove $s_{1,0}$ is the initial state i_1 of $\mathcal{P}_1$. $s_0 \cap F_1 = (i_{1,0} \cup i_{2,0}) \cap F_1 = (i_{1,0} \cap F_1) \cup (i_{2,0} \cap F_1) = (i_{1,0} \cap F_1) = s_{1,0}$.
2. Prove $\forall k$ a_k is applicable in $s_{1,k-1}$. Since the precondition of a_k in $\mathcal{P}_1 \otimes \mathcal{P}_2$ is the conjunction of the precondition of a_k in $\mathcal{P}_1$ with another formula, the precondition of a_k in $\mathcal{P}_1$ is satisfied by s_{k-1}. Furthermore, s_{k-1} is identical to $s_{1,k-1}$ except for some facts from F_2 which are irrelevant. Therefore, the precondition is satisfied.
3. Prove $\forall k$, $a_k[s_{1,k-1}] = s_{1,k}$. Again, we see that the conditional effects of a_k that modify the truth value of the facts in F_1 are the same for both problems $\mathcal{P}_1 \otimes \mathcal{P}_2$ and $\mathcal{P}_1$. Therefore, the state reached after applying the effects is the same: $s_{k-1}[a_k] \cap F_1 = (s_{k-1} \cap F_1)[a_k]$.
4. Prove $s_{1,n}$ is a goal state of $\mathcal{P}_1$. This is similar to Point 2 on precondition.

□

It is easy to extend Lemma 4 to multiple problems: $\Pi(\mathcal{P}_1 \otimes \cdots \otimes \mathcal{P}_n) = \Pi(\mathcal{P}_1) \cap \cdots \cap \Pi(\mathcal{P}_n)$.

Definition 18. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two* trivial *classical planning problems with $F_1 \cap F_2 = \emptyset$ and $\{name(a) \mid a \in A_1\} = \{name(a) \mid a \in A_2\}$. The* disjunction *of $\mathcal{P}_1$ and $\mathcal{P}_2$ is the classical planning problem $\mathcal{P} = \mathcal{P}_1 \oplus \mathcal{P}_2 = \langle F, A, I, G \rangle$ where*

- *$F = F_1 \cup F_2$, $i = i_1 \cup i_2$, $G = G_1 \vee G_2$, and*
- *$A = \{a_1 \oplus a_2 \mid a_1 \in A_1 \wedge a_2 \in A_2 \wedge name(a_1) = name(a_2)\}$ where $a = a_1 \oplus a_2$ is defined by*
  - *$name(a) = name(a_1)$, and*
  - *$\mathit{eff}(a) = \mathit{eff}(a_1) \cup \mathit{eff}(a_2)$.*

Unlike the conjunction, the disjunction requires an additional condition in order to get the desired behavior.

Lemma 5. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two classical planning problems. If $\mathcal{P}_1$ and $\mathcal{P}_2$ are such that their action preconditions are trivial ($\forall j \in \{1,2\}.\ \forall a \in A_j.\ pre(a) = \textbf{\textit{True}}$), then the following holds:* $\Pi(\mathcal{P}_1 \oplus \mathcal{P}_2) = \Pi(\mathcal{P}_1) \cup \Pi(\mathcal{P}_2)$

Proof of Lemma 5. Let $\pi = a_1, \cdots, a_n$ and π is valid in $\mathcal{P}_1$: $s_{1,0} \xrightarrow{a_1} s_{1,1} \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_{1,n}$. π is also valid in $\mathcal{P}_2$: $s_{2,0} \xrightarrow{a_1} s_{2,1} \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_{2,n}$. The corresponding actions a_i in $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_1 \oplus \mathcal{P}_2$ share the same action name by definition.

The proof is very similar to that of Lemma 4. However, we need to make the following changes.

We first assume that the plan is valid for one of the problems (but still assume that the states $s_{q,j}$ are the result of applying the actions in the other problem even when they are not applicable). One can easily check that the four conditions (Part A in Proof 4.5) are satisfied.

We then assume the plan of $\mathcal{P}_1 \oplus \mathcal{P}_2$ leads to a goal state, and we need to check that the plan is valid for one of the problems $\mathcal{P}_1$ and $\mathcal{P}_2$. Because the goal condition is the disjunction $G_1 \vee G_2$ where G_1 and G_2 pertain to two different sets of facts F_1 and F_2, it is immediate that $s_n \cap F_p$ satisfies G_p for some p. Wlog, we assume $p = 1$.

Similar to Lemma 4, Conditions 1 and 3 are easy to verify. Condition 2 is trivial because the preconditions are trivial. Condition 4 is satisfied by the choice of p. □

It is easy to extend Lemma 5 to multiple projected problems: $\Pi(\mathcal{P}'_1 \oplus \cdots \oplus \mathcal{P}'_n) = \Pi(\mathcal{P}'_1) \cup \cdots \cup \Pi(\mathcal{P}'_n)$.

Given a set T of CTS, we demonstrate how to generate a new planning problem (composite problem) such that all valid plans are exactly those that the counter-tag sets do not serve as counter-examples to:

$$P_B = P_{T_1} \otimes \cdots \otimes P_{T_n}$$

where $B = \{T_1, \ldots, T_n\}$ and

$$P_{T_i} = P_{c_1} \oplus \cdots \oplus P_{c_m}$$

where $T_k = \{c_1, \ldots, c_m\}$ and P_{c_k} is the trivialization of the projection of a PCP problem P over a context c_k. The composite problem is a classical planning problem used for computing candidate plans.

We now describe the implementation of the function `compute_candidate_plan` in Line 10 of Algorithm 6. The function proceeds by iterating over each CTS in the set B. For each counter-tag c_i in a given CTS, we construct its corresponding trivial planning problem P_{c_i}. These trivial problems are then combined using disjunction to form a disjunctive problem for the CTS: $P_{T_i} = P_{c_1} \oplus \cdots \oplus P_{c_m}$. After computing the disjunctive problem for each CTS, we aggregate them using conjunction to obtain the composite problem: $P_B = P_{T_1} \otimes \cdots \otimes P_{T_n}$. This composite problem is a classical abstraction, and a classical planner is used to solve the abstracted problem. If the classical planner finds a solution to P_B, the resulting plan is returned as the candidate plan. Otherwise, if P_B is unsolvable, the function returns $\perp$.

Example 22. *Let us use the example in Table 4.1 to explain the details of our method.*

1. *In the first iteration, CTS has two counter-tags: x_1 and x_3, so the disjuncted problem $P_1 = P_{x1} \oplus P_{x3}$:*
   - *initial condition:* $[(x_1\ int1) \wedge (valid\ int1)] \wedge [(x_3\ int3) \wedge (valid\ int3)]$
   - *goal condition:* $[(x_2\ int1) \wedge (valid\ int1)] \vee [(x_2\ int3) \wedge (valid\ int3)]$

   Since this is the first iteration, the conjuncted problem is P_1 itself: $P_{B1} = P_1$.
2. *In the second iteration, CTS is updated as $\{x_2\}$, so the disjuncted problem is $P_2 = P_{x_2}$, in which:*
   - *initial condition:* $[(x_2\ int2) \wedge (valid\ int2)]$
   - *goal condition:* $[(x_2\ int2) \wedge (valid\ int2)]$

   The conjuncted problem is $P_{B2} = P_1 \otimes P_2$, in which
   - *initial condition:* $[(x_1\ int1) \wedge (valid\ int1)] \wedge [(x_2\ int2) \wedge (valid\ int2)] \wedge [(x_3\ int3) \wedge (valid\ int3)]$
   - *goal condition:* $\{[(x_2\ int1) \wedge (valid\ int1)] \vee [(x_2\ int3) \wedge (valid\ int3)]\} \wedge \{[(x_2\ int2) \wedge (valid\ int2)]\}$

3. *In the third iteration, CTS has one counter-tag y_2, so in the disjuncted problem $P_3 = P_{y2}$,*

   - *initial condition:* $[(y_2\ int5) \wedge (valid\ int5)]$
   - *goal condition:* $[(y_2\ int5) \wedge (valid\ int5)]$

   The conjuncted problem is updated to $P_{B3} = P_1 \otimes P_2 \otimes P_3$, in which

   - *initial condition:* $[(x_1\ int1) \wedge (valid\ int1)] \wedge [(x_2\ int2) \wedge (valid\ int2)] \wedge [(x_3\ int3) \wedge (valid\ int3)] \wedge [(y_2\ int5) \wedge (valid\ int5)]$
   - *goal condition:* $\{[(x_2\ int1) \wedge (valid\ int1)] \vee [(x_2\ int3) \wedge (valid\ int3)]\} \wedge \{[(x_2\ int2) \wedge (valid\ int2)]\} \wedge \{[(y_2\ int5) \wedge (valid\ int5)]\}$

4. *In the forth iteration, CTS has two counter-tags x_1 and y_1, so the disjuncted problem $P_4 = P_{x1} \oplus P_{y_1}$ should be:*

   - *initial condition:* $[(x_1\ int1) \wedge (valid\ int1)] \wedge [(y_1\ int4) \wedge (valid\ int4)]$
   - *goal condition:* $[(x_2\ int1) \wedge (valid\ int1)] \vee [(y_2\ int4) \wedge (valid\ int4)]$

   Then the conjuncted problem is $P_{B4} = P_1 \otimes P_2 \otimes P_3 \otimes P_4$ in which

   - *initial condition:* $[(x_1\ int1) \wedge (valid\ int1)] \wedge [(x_2\ int2) \wedge (valid\ int2)] \wedge [(x_3\ int3) \wedge (valid\ int3)] \wedge [(y_1\ int4) \wedge (valid\ int4)] \wedge [(y_2\ int5) \wedge (valid\ int5)]$
   - *goal condition:* $\{[(x_2\ int1) \wedge (valid\ int1)] \vee [(x_2\ int3) \wedge (valid\ int3)]\} \wedge \{[(x_2\ int2) \wedge (valid\ int2)]\} \wedge \{[(y_2\ int5) \wedge (valid\ int5)]\} \wedge \{[(x_2\ int1) \wedge (valid\ int1)] \vee [(y_2\ int4) \wedge (valid\ int4)]\}$

As you can see, in each conjunct, only one of the sub-goals needs to be achieved. This mechanism effectively avoids initial states (or combinations of initial states) that would have no solution. If a particular sub-goal (or a combination of sub-goals) cannot be achieved, the classical planner can opt to pursue another sub-goal (or a different combination of sub-goals), ensuring the problem remains solvable.

4.6 Hitting Set Strategy

After assembling the sub-problems, our general approach involves computing candidate plans from the set $\Pi(P_{T_1}) \cap \cdots \cap \Pi(P_{T_n})$ where $\Pi(P_{T_i}) = \Pi(P_{c_1}) \cup \cdots \cup \Pi(P_{c_m})$.

Algorithm 8 `p-CPCES-hit`

```
1: Input: PCP problem 𝒫
2: Output: a plan π for 𝒫, or UNSAT
3: B := ∅                                          ▷ a set of counter-tags
4: π := ε                                          ▷ candidate plan
5: loop
6:     while π = ⊥ do
7:         H := unique_hit(B)
8:         if H = ⊥ then
9:             return no plan
10:        π := compute_candidate_plan(𝒫, H)
11:    CTS := compute_CTS(𝒫, π)
12:    if CTS = ⊥ then
13:        return π
14:    B := B ∪ {CTS}
```

This ensures that the plan satisfies at least one of the counter-tags in each iteration. However, classical planners often struggle with solving problems that have disjunctive goal conditions. To address this, we shift the choice outside the planner: for every counter-tag set, we pre-select one counter-tag in advance. The collection of these selected counter-tags forms a hitting set $H = \{h_1, \ldots, h_n\}$. The candidate plan is then computed for the conjunctive problem $\Pi(P_H) = \Pi(P_{1h_1}) \cap \cdots \cap \Pi(P_{nh_n})$. In this way, the disjunctive structure is eliminated, and the planner only needs to solve a purely conjunctive goal condition, which greatly improves efficiency.

Any candidate plan will then belong to the set $\Pi(P_H) = \Pi(P_{1h_1}) \cap \cdots \cap \Pi(P_{nh_n})$, where for each j, $h_j \in \{1, \ldots, k_j\}$, meaning $H = \{h_1, \ldots, h_n\}$ forms a hitting set (Slaney 2014) of the counter-tag sets B. If H were known in advance, finding the plan would be faster, as it would involve only conjunctive goals. To leverage this, we propose searching through the hitting sets of B as described in Algorithm 8. We name this algorithm as `p-CPCES-hit`.

Compared with `p-CPCES`, finding a candidate plan in `p-CPCES-hit` involves an additional loop (Line 6) to iterate through the hitting sets. The method `unique_hit` in Line 7 produces a different hitting set in each iteration. If none of the hitting sets leads to a plan (as indicated in Line 7 when all hitting sets have been

exhausted), then no valid plan exists. The function `compute_candidate_plan` in Algorithm 8 is different from the one in Algorithm 6. In Algorithm 8, the relevant counter-tags that need to be satisfied in the candidate plan have already been identified via the hitting set H. We construct a trivial planning problem P_{h_i} for each counter-tag $h \in H$, and then assemble these trivial planning problems into a composite problem via conjunction: $P_B = P_{h_1} \otimes \cdots \otimes P_{h_n}$. Then, a classical planner is used to solve P_B. If the classical planner finds a solution to P_B, the resulting plan is returned as the candidate plan. Otherwise, if P_B is unsolvable, the function returns $\perp$. It is important to note that neither `p-CPCES` nor `p-CPCES-hit` aim to find an optimal plan.

4.7 Experimental Results

We evaluated our methods across four probability thresholds: $\tau \in \{0.99, 0.90, 0.75, 0.5\}$. The classical planners utilized in our implementation were `FF` (Hoffmann 2001) and `Madagascar` (Rintanen 2014)[1]. We explored two hitting set strategies within `p-CPCES-hit`: minimal hitting sets and random hitting sets. Additionally, our implementations were compared against `P-FF` (Domshlak and Hoffmann 2006, 2007). Our evaluation was conducted over most of the domain and problem set used in ICAPS-24 (Zhang et al. 2020) (used in Chapter 3, originally taken from IPC 2008), and we add some new domains and problems. For each instance used in ICAPS-24, we assign probabilities to each set of uncertain initial facts using a uniform probability distribution. For example, when solving the problem of a robot with an uncertain initial position in a 3×4 grid and tasked with reaching a designated location, we assign a probability of 0.3333 to each x-coordinate and 0.25 to each y-coordinate. The benchmark suite includes seven domains: `BOMB`, `COINS`, `DISPOSE`, `ONEDISPOSE`, `LOGISTICS`, `LOOK-GRAB`, and `UTS`. Since `p-CPCES` and `p-CPCES-hit` involve random selection of counter-tags in each iteration, each instance was run nine times to mitigate the impact of randomness, and the median results were reported. We also compared our algorithm with the state-of-the-art PCP planner `P-FF` (Domshlak

[1]We ran algorithm C with $R = 1.2$, checking horizons from 0 to 50, using $\exists$-step -semantics.

and Hoffmann 2006, 2007). For each instance, `P-FF` was executed only once, as there is no randomness in `P-FF` 's searching process.

In our implementation, minimal hitting sets generated by Hitman Module (pysat.examples.hitman) (Ignatiev et al. 2018) in Python with default settings. We also experimented with a variant of our algorithm that uses an arbitrary, possibly non-minimal, hitting set. We call these "random" hitting sets, which are computed by converting the hitting set problem to an equivalent SAT problem that we solve using the Kissat algorithm (Biere et al. 2024). Each experiment was limited to 1800 seconds. All experiments were conducted on a computing system equipped with an AMD Ryzen Threadripper 3990X 64-Core Processor and 128GB of RAM. All experimental results are listed from Table 1 to Table 6.

The following notations are used throughout:

- `p-CPCES-hit-minimal`: `p-CPCES-hit` using `FF` as classical planner with the minimal hitting set strategy.
- `p-CPCES-hit-random`: `p-CPCES-hit` using `FF` as classical planner with the random hitting set strategy.
- `FF-p-CPCES`: `FF` planner used in `p-CPCES`.
- `FF-Hit-Minimal`: `FF` planner used in `p-CPCES-hit-minimal`.
- `FF-Hit-Random`: `FF` planner used in `p-CPCES-hit-random`.
- `Mad-p-CPCES`: `Madagascar` planner used in `p-CPCES`.
- `Mad-Hit-Minimal`: `Madagascar` planner used in `p-CPCES-hit-minimal`.
- `Mad-Hit-Random`: `Madagascar` planner used in `p-CPCES-hit-random`.

4.7.1 p-CPCES vs. p-CPCES-hit-minimal

Both methods performed best when $\tau = 0.99$, as the counter-tag probability threshold per iteration was only 0.01. This low threshold offered two major advantages:

- Only a small number of counter-tags are required, as the probability of the initial states represented by these counter-tags is sufficient to meet the spec-

ified criteria. This significantly reduces the computational time needed to generate the counter-tags.

- Since the number of counter-tags in each iteration is small, the disjunctive clauses in the goal state is short. This enables the classical planner to solve the problem more efficiently.

For example, in domains like `DISPOSE` p8-1 and `UTS` p7, only one counter-tag was required per iteration. This makes the goal condition be a conjunctive goal, avoiding the disjunctive goals appear in the goal condition, significantly reducing runtime.

However, performance deteriorated for both `p-CPCES` and `p-CPCES-hit-minimal` while τ decreasing. For instance, `DISPOSE` p4-2 solved in 9.48 seconds with $\tau = 0.99$ using `FF-Hit-Minimal`, took 368.75 seconds with $\tau = 0.9$, and failed to solve within 1800 seconds when $\tau \leq 0.75$. This decline is attributed to the challenge of building a description of counter-examples with sufficient probability mass, which requires numerous counter-tags and results in lengthy goal descriptions. In practice, planning systems struggle with problems that involve long and disjunctive goal conditions, making such formulations particularly difficult to handle.

With both `FF` and `Madagascar` planners, `p-CPCES-hit-minimal` consistently outperformed `p-CPCES` when $\tau \leq 0.9$. This advantage stemmed from the hitting set method, which guided the classical planner by pre-selecting tags to be satisfied. For $\tau = 0.99$, this advantage diminished as most problems required only one counter-tag per iteration, resulting in conjunctive goal conditions.

For high τ values, `FF-p-CPCES` consistently outperformed `Mad-p-CPCES` (Figure 4.2a). However, for low τ, `Madagascar` proved superior (Figure 4.2c), solving problems that `FF` could not within the 1800-second limit. We found that `FF` was the fastest base planner in problems with short conjunctive goals, whereas `Madagascar` was the fastest in case of lengthy goals with disjunctive conditions such as occur for low values of τ.

Similar to the performance trends observed with `FF-p-CPCES`, `FF-Hit-Minimal` significantly outperforms `Mad-Hit-Minimal` when $\tau = 0.99$ or 0.9 (Figure 4.3a and Figure 4.3b). However, for lower probability thresholds, `Mad-Hit-Minimal`

solves more problems compared to `FF-Hit-Minimal`. For instance, when $\tau = 0.75$, out of 73 problems, `Mad-Hit-Minimal` successfully solves 6 more problems, primarily in `LOGISTICS` problems. Apart from these 6 problems, their performance is nearly identical on the remaining problems. Despite this, when considering the overall number of problems solved across all τ values, `FF-Hit-Minimal` consistently outperforms `Mad-Hit-Minimal`. This broader effectiveness is evident in Figure 4.3e, where `FF-Hit-Minimal` solves more problems in total.

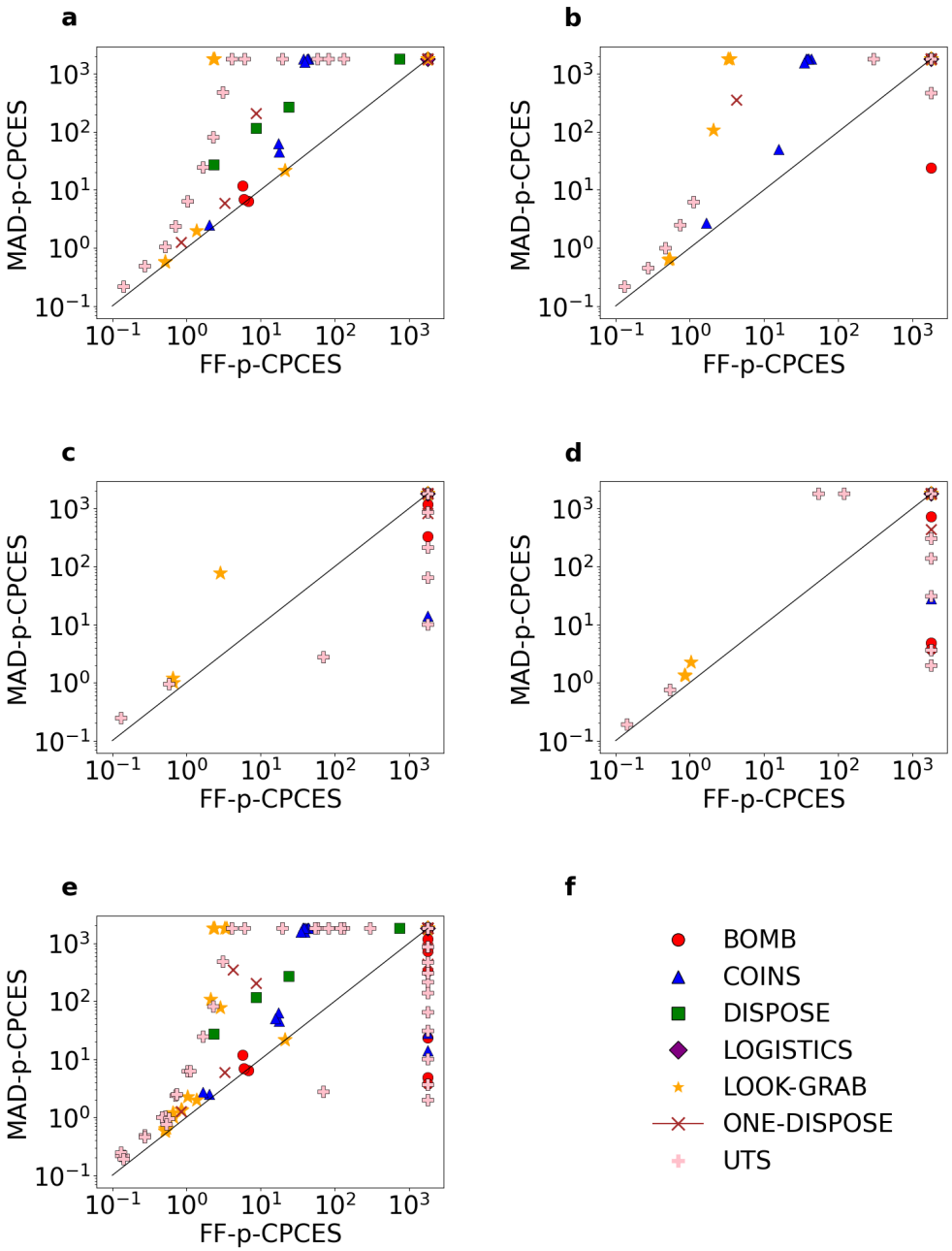


Figure 4.2: Runtime required to find a valid plan using `FF-p-CPCES` and `Mad-p-CPCES`. Figure a shows the case when the probability threshold $\tau = 0.99$. Figure b corresponds to $\tau = 0.90$. Figure c corresponds to $\tau = 0.75$. Figure d corresponds to $\tau = 0.50$. Figure e combines all the problems. Figure f presents the legend.

4.7.2 p-CPCES-hit-minimal vs. p-CPCES-hit-random

As τ decreased, p-CPCES-hit-minimal struggled more, whereas p-CPCES-hit-random had no clear dependence on τ. For example, FF-Hit-Random solved BOMB p20-1 fastest with $\tau = 0.5$ but slowest with $\tau = 0.9$, while COINS p12 was fastest at $\tau = 0.99$ and slowest at $\tau = 0.75$. This variability results from the random nature of hitting set generation, which occasionally produces large sets meeting the probability threshold early. Overall we found p-CPCES-hit-random consistently outperformed p-CPCES-hit-minimal, because:

- Generating random hitting sets was computationally cheaper than minimal ones, particularly in later iterations.
- In our problem set using random hitting sets achieves a valid plans with fewer iterations. This is especially the case for small τ, because larger hitting sets increased the probability of success of the computed plan.

Figure 4.5 compare the number of iterations of FF-Hit-Random and FF-Hit-Minimal for problems that both algorithms successfully solve. Notably, across all domains except BOMB and LOOK-GRAB, FF-Hit-Random consistently requires fewer iterations to find a valid plan. This highlights the advantage of FF-Hit-Random's random generation of hitting sets, which often produces sufficiently large hitting sets early in the process. As a result, it achieves a valid plan with fewer iterations compared to FF-Hit-Minimal. From Figure 4.4, we can observe that there is no significant difference in the plan length between FF-Hit-Random and FF-Hit-Minimal. Observations analogous to the above also occur in the case of variants of our algorithm using Madagascar as a base planner.

We observed an interesting phenomenon in the BOMB domain with 100 bombs: when $\tau = 0.5$, FF-Hit-Random was able to find a valid plan within the time limit, but when τ exceeded 0.5, it consistently failed to do so. Upon further analysis, we found that at $\tau = 0.5$, FF-Hit-Random required only three iterations to find a valid plan. However, when τ was greater than 0.5, even after computing dozens of iterations, valid plan has not been found yet. This behavior can be explained by the feature of our SAT solver Kissat. At $\tau = 0.5$, the goal condition clauses

contain many disjuncts, making the problem more complex. In such cases, Kissat tends to generate unnecessarily long hitting sets. These long hitting sets are more likely to cover a substantial portion of the initial states' probabilities, allowing a valid plan to be found in the early iterations.

4.7.3 P-FF vs. p-CPCES-hit

Previous studies indicate P-FF performs well for lower τ, with declining performance as τ increases (Domshlak and Hoffmann 2007). However, in our benchmarks, this trend was not consistent across all domains. For instance, ONEDISPOSE required 381 seconds with $\tau = 0.99$ but only 0.14 seconds with $\tau = 0.5$. Conversely, COINS p16 became harder for P-FF as τ decreased to 0.5. These experimental results demonstrate that P-FF 's performance is influenced not only by the probability threshold τ but also by the specific characteristics of the problem (Figure 4.6).

When comparing FF-Hit-Random with P-FF (Figure 4.7), P-FF demonstrates better performance for simpler problems that can be solved within 10 seconds. However, for more complex problems requiring over 10 seconds to solve, FF-Hit-Random outperforms P-FF. Interestingly, at $\tau = 0.5$, P-FF surpasses FF-Hit-Random in the number of problems it solves within 100 seconds (Figure 4.7d), unlike the higher thresholds ($\tau = 0.99 - 0.75$), where P-FF's advantage is primarily observed only within the 10-second range. These findings highlight the suitability of p-CPCES-hit for addressing more complex problems effectively. For BOMB and LOGISTICS domains, P-FF consistently outperformed p-CPCES-hit, regardless of τ. This suggests P-FF's heuristics are better suited to these domain structures.

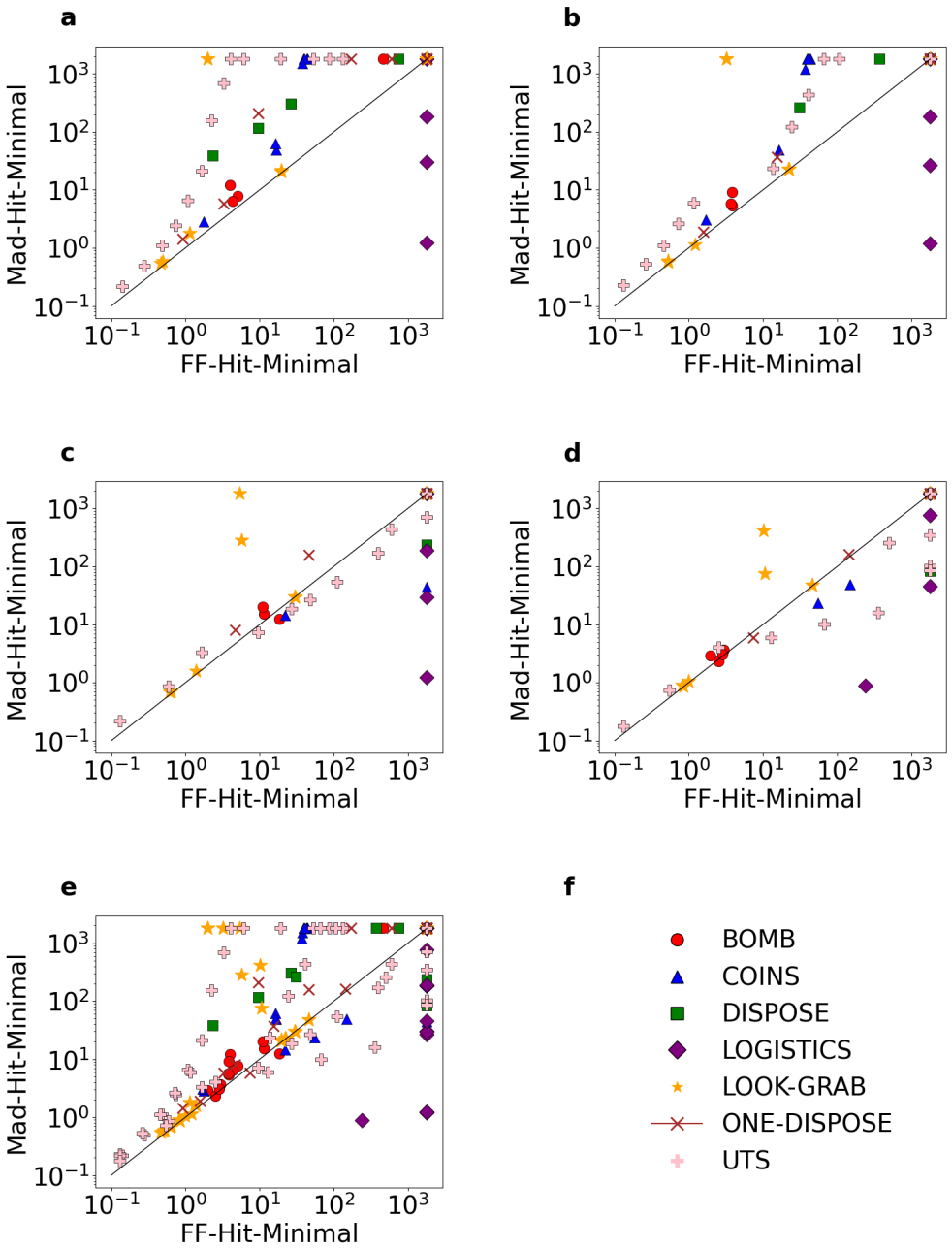


Figure 4.3: Runtime required to find a valid plan using `FF-Hit-Minimal` and `Mad-Hit-Minimal`. Figure a shows the case when the probability threshold $\tau = 0.99$. Figure b corresponds to $\tau = 0.90$. Figure c corresponds to $\tau = 0.75$. Figure d corresponds to $\tau = 0.50$. Figure e combines all the problems. Figure f presents the legend.

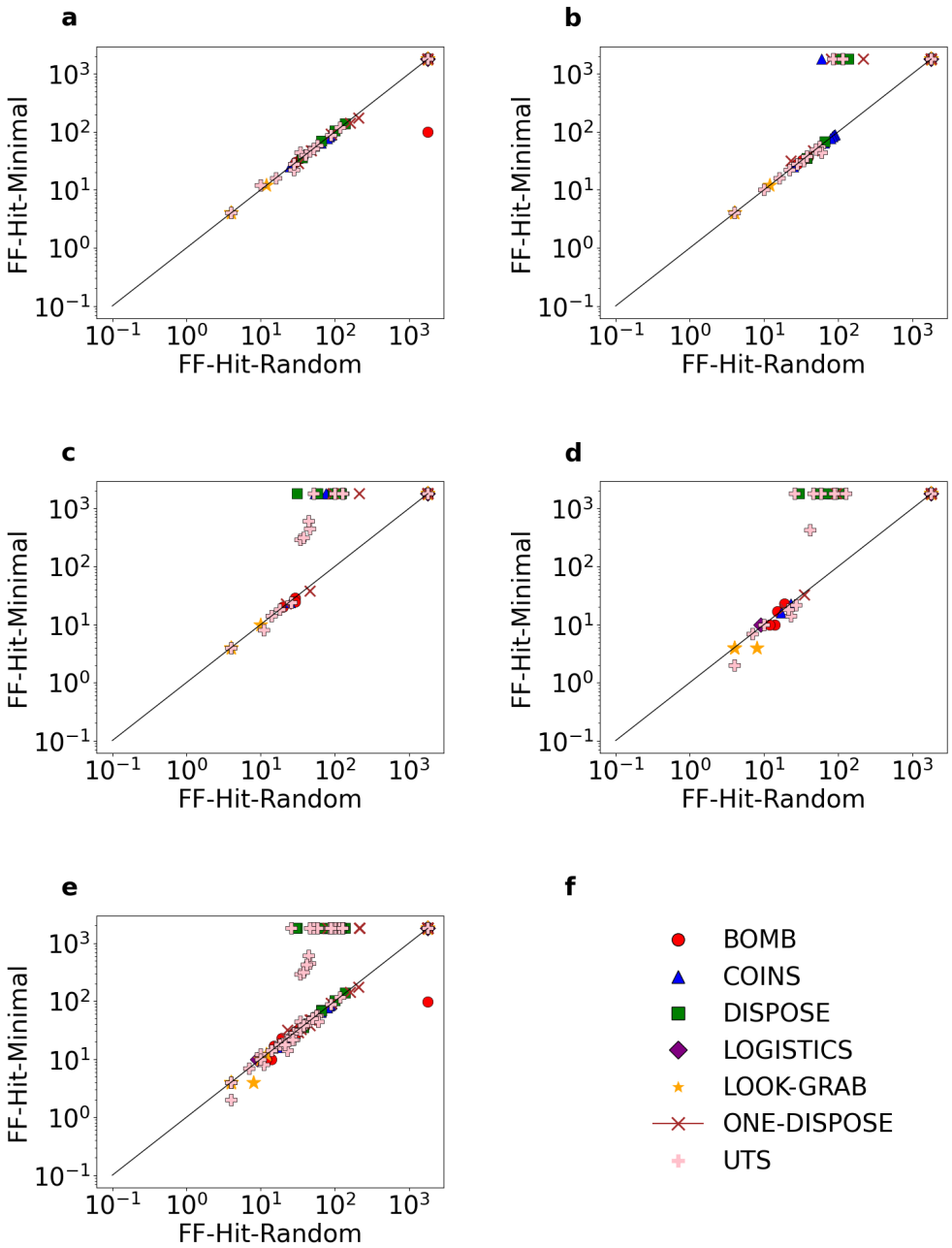


Figure 4.4: Comparison of the plan length between `FF-Hit-Random` and `FF-Hit-Minimal`. Figure a shows the case when the probability threshold $\tau = 0.99$. Figure b corresponds to $\tau = 0.90$. Figure c corresponds to $\tau = 0.75$. Figure d corresponds to $\tau = 0.50$. Figure e combines all the problems. Figure f presents the legend.

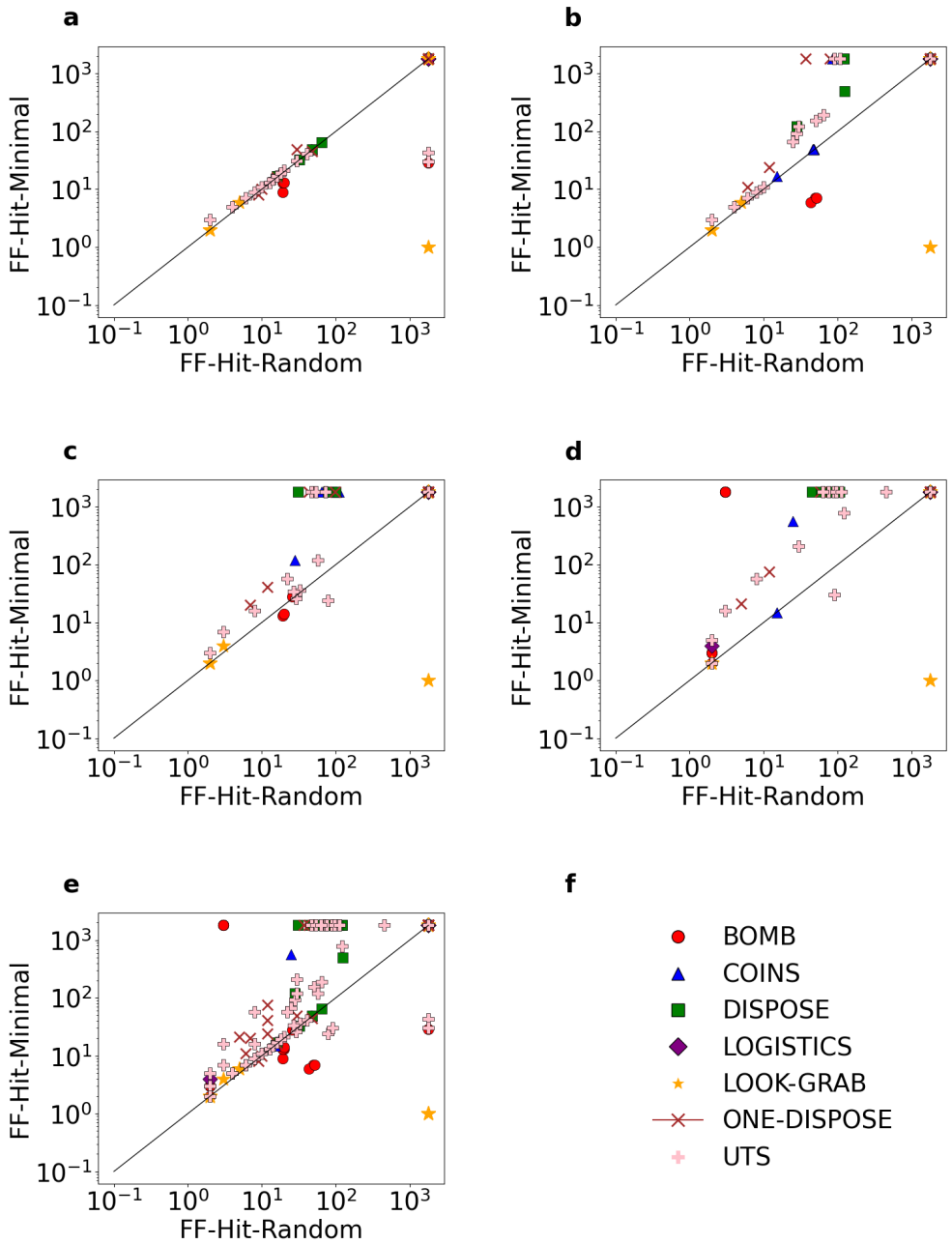


Figure 4.5: Comparison of the number of iterations required by `FF-Hit-Random` and `FF-Hit-Minimal`. Figure a shows the case when the probability threshold $\tau = 0.99$. Figure b corresponds to $\tau = 0.90$. Figure c corresponds to $\tau = 0.75$. Figure d corresponds to $\tau = 0.50$. Figure e combines all the problems. Figure f presents the legend.

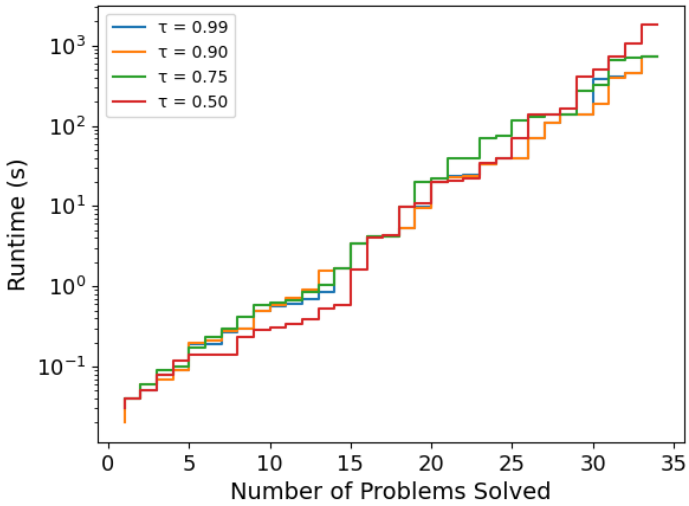


Figure 4.6: Number of problems solved by `P-FF` at various probability thresholds τ.

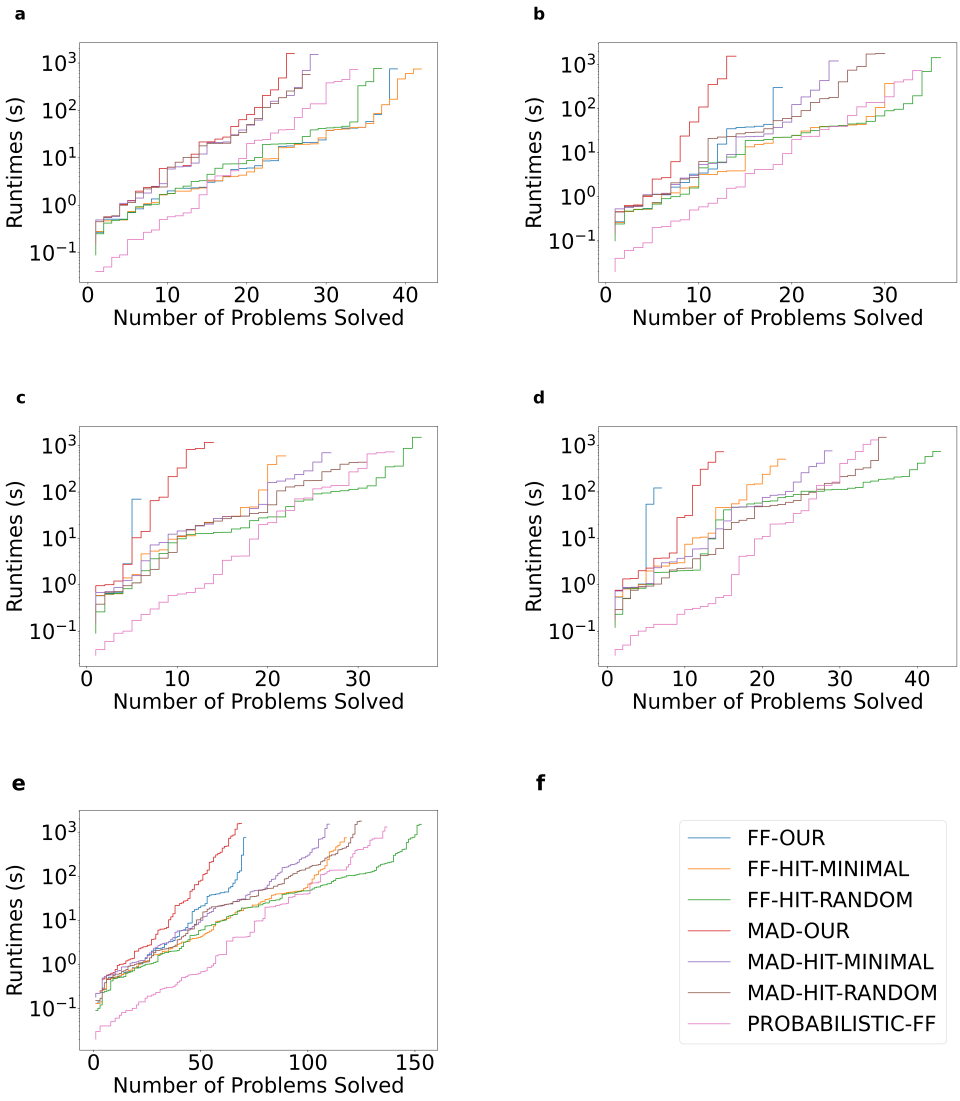


Figure 4.7: Number of problems solved by various algorithms. Figure a shows the case when the probability threshold $\tau = 0.99$. Figure b corresponds to $\tau = 0.90$. Figure c corresponds to $\tau = 0.75$. Figure d corresponds to $\tau = 0.50$. Figure e combines all the problems. Figure f presents the legend.

4.8 Conclusions

This chapter presents a counter-example based approach to solving PCP problems. Our proposed method, `p-CPCES`, projects the original PCP problem onto a set of contexts and uses counter-tags to represent sets of counter-examples. The probability of each counter-tag is computed using d-DNNF compilation. The `p-CPCES` algorithm iteratively constructs and solves composite classical planning problems derived from the counter-tags. In each iteration, a candidate plan is computed for the current composite problem. If no plan can be found, the PCP problem is determined to be unsolvable. Otherwise, the algorithm evaluates the counter-tags of the candidate plan and computes their probability using d-DNNF. If all counter-tags are identified but the total probability mass of initial states corresponding to these counter-tags is less than $1 - \tau$, the last candidate plan is accepted as a solution to the PCP problem. Due to the disjunctive nature of goals in the composite problems generated by `p-CPCES`, we further develop `p-CPCES-hit`, which integrates a hitting set strategy to assist classical planners in selecting subgoals.

Experimental results demonstrate that both `p-CPCES` and `p-CPCES-hit` effectively solve PCP problems, with `p-CPCES-hit` demonstrating significantly improved performance. Our experiments evaluate performance across multiple probability thresholds, revealing that both methods are particularly effective when the probability threshold is high. We also compare the performance when using different classical planners — `FF` and `Madagascar`— within our approaches. We find that when the probability threshold is high, `FF` performs better, whereas `Madagascar` is faster under low probability thresholds. Additionally, we explore the impact of different hitting set strategies in `p-CPCES-hit`, comparing minimal and random hitting sets. We observe that the use of random hitting sets leads to higher computational efficiency. Finally, we compare `p-CPCES-hit` with the current state-of-the-art PCP planner, `P-FF`. The results show that while `P-FF` excels at solving simpler PCP problems (solved within 10 seconds), `p-CPCES-hit` (using `FF` as the classical planner) outperforms `P-FF` on more challenging problems that require more than 10 seconds to solve.

Chapter 5

Parallelizing Probabilistic Conformant Planning

In the previous chapter, we introduced the `p-CPCES-hit` in Algorithm 8, which consists of three core components:

1. **Hitting Set Calculation** (Line 7): In each iteration of the inner loop (Line 6), `p-CPCES-hit` computes a unique hitting set from the counter-tag set.
2. **Candidate Plan Searching** (Line 10): Using the hitting set, `p-CPCES-hit` constructs a classical planning problem and searches a candidate plan.
3. **Counter-Tag Set Computation** (Line 11): The candidate plan is then used to compute the counter-tag set.

These three components are interdependent, with each step needing to complete before the next can begin. Any of these steps can become a bottleneck in the algorithm. For example, in the `BOMB` p100-1 problem, computing the counter-tag set is the bottleneck because up to 100 tags need to be validated. In contrast, for the `DISPOSE` p8-3 problem, searching the candidate plan is the bottleneck due to the complexity of solving each classical planning problem. This observation motivated us to explore strategies to mitigate bottlenecks by pursuing multiple independent searches in parallel.

In this chapter, we introduce `parallel-CPCES`, an extension of `p-CPCES-hit` that leverages parallel computation to calculate multiple candidate plans in each iteration, effectively mitigating bottlenecks in any single component. To distinguish

p-CPCES-hit from parallel-CPCES, we also refer to p-CPCES-hit as the serial algorithm. The running example used in this chapter is from 1.1.5.

5.1 An Overview of parallel-CPCES

parallel-CPCES leverages parallelism by distributing computational tasks across multiple workers for each component. Its core idea is to compute multiple candidate plans concurrently during each iteration. This approach requires generating multiple hitting sets per iteration and utilizing multiple workers to calculate the counter-tag set for each candidate plan. Below, we discuss the parallelization strategies for each component:

1. **Multiple Hitting Sets Calculation**: In each iteration, unlike the serial algorithm, which calculates the next hitting set only after completing the search for a plan, parallel-CPCES employs a dedicated worker exclusively for generating hitting sets. Our parallel system poses multiple independent classical abstractions at each iteration. As a result, processes assigned to planning can pursue candidate plans in parallel with processes assigned to the computation of hitting sets. This overall scheme exhibits the conditions necessary to have hitting sets and planning work continuously available to pursue. This worker operates independently of the plan results and iteratively computes hitting sets in a continuous loop. This creates the conditions necessary for computing multiple plans in each iteration concurrently.

2. **Parallel Candidate Plan Computation**: Using the multiple hitting sets from step one, parallel-CPCES assigns multiple workers to compute candidate plans in parallel. Since the difficulty of solving classical planning problems varies, the time required for each worker to compute a plan may differ. parallel-CPCES advances to the next iteration as soon as the first candidate plan is computed. This mechanism ensures that each iteration identifies a candidate plan as quickly as possible, expediting progress to the next iteration. Importantly, while the main process moves forward, the remaining workers continue computing plans for the current iteration. This is because any of these plans might turn out to be the valid plan for the PCP

problem.

3. **Parallel Counter-Tag Set Computation**: Only the first computed plan at iteration i is selected as the candidate plan for the iteration $i+1$. Nonetheless, `parallel-CPCES` concurrently poses and solves multiple classical planning problems at each iteration i, any of which may correspond to a solution to the underlying PCP. The parallel system employs multiple workers to compute counter-tag sets for all available plans in parallel. If a plan is found to lack sufficient counter-tags, it is identified as the valid plan for the PCP problem and the program terminates having successfully solved the PCP. In practice this mechanism is the basis of a parallel system that is able to find a valid plan more quickly than the serial procedure in Algorithm 8.

Let us use the example from Section 1.1.5 to illustrate how `parallel-CPCES` works. This example focuses on the primary mechanisms of `parallel-CPCES` to help readers understand its layout. More intricate designs for each component will be discussed in subsequent sections.

Example 23. *This example, illustrated in Figure 5.1, demonstrates how* ***parallel-CPCES*** *operates. In this example, we have three workers* $\{p_1, p_2, p_3\}$ *for plan computation, one worker* c *for computing CTS, and one worker* h *for computing hitting sets. For simplicity, we assume that computing each CTS takes 1 second, while the time for computing hitting sets is negligible (0 seconds). The entire process takes 10 seconds to find the final solution, so we denote each time point as* $\{t_0 \dots t_{10}\}$*, where* t_0 *marks the start and* t_{10} *marks the end of the program. In the figure, each hitting set is represented by a box containing 10 adjacent squares, where black and red dots indicate processing tasks. Black dots represent the time taken to compute a plan from a hitting set, while red dots represent the time taken to compute CTS for the corresponding plan. For example, hitting set* H_1 *shows a black dot in the second square and a red dot in the third square, meaning that plan computation from* $H1$ *occurs from* t_1 *to* t_2*, and CTS computation from the resulting plan occurs from* t_2 *to* t_3*. Below is a second-by-second breakdown of the process.*

- t_0*:*
    1. ***parallel-CPCES*** *starts computation with iteration* $i = 0$ *and candidate*

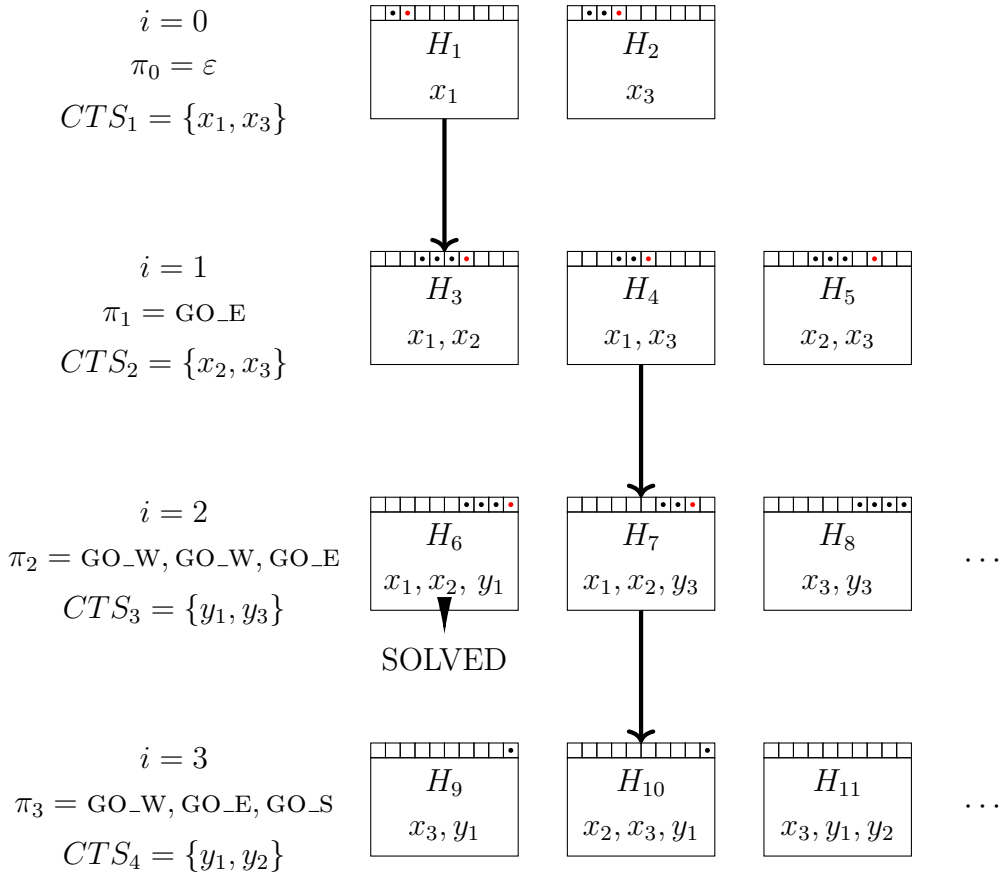


Figure 5.1: Illustration of how parallel-CPCES operates. Each large box represents a classical abstraction of the PCP problem, parameterized by a hitting set. The array of cells above each box shows a linear timeline: a black token indicates a classical planner scheduled to solve the problem at that time; a red token indicates a counter-tag set calculation based on a plan generated by the planner.

plan $\pi_0 = \varepsilon$.

- $t_0 - t_1$*:*
  1. *Worker* c *computes* $CTS_1 = \{x_1, x_3\}$ *using* π_0.
  2. *worker* h *generates two hitting sets:* H_1 *and* H_2.
- $t_1 - t_2$*:*
  1. *Worker* p_1 *computes plan* $\pi_{H1} = \text{GO\_E}$ *from* H_1 *in 1 second.*
  2. *Worker* p_2 *starts computing a plan from* H_2 *but does not find one within 1 second.*
  3. *Since* π_{H1} *is the first plan found, it becomes the candidate plan for this iteration, updating* $\pi_1 = \text{GO\_E}$.
- $t_2 - t_3$
  1. *Worker* c *computes* $CTS_2 = \{x_2, x_3\}$ *using* π_1.
  2. *The main process advances to iteration* $i = 1$.
  3. *Worker* h *generates three new hitting sets:* H_3, H_4, *and* H_5.
  4. *Worker* p_2 *completes plan computation for* H_2, *producing* π_{H2}.
- $t_3 - t_4$
  1. p_1, p_2, *and* p_3 *compute plans for* H_3, H_4, *and* H_5, *respectively.*
  2. *Worker* c *evaluates* π_{H2} *and determines it is not a valid plan.*
- $t_4 - t_5$
  1. *Worker* p_2 *computes* $\pi_{H4} = \text{GO\_W}, \text{GO\_W}, \text{GO\_E}$.
  2. *Since* π_{H4} *is the first plan found in this iteration, it becomes the candidate plan, updating* $\pi_2 = \text{GO\_W}, \text{GO\_W}, \text{GO\_E}$.
- $t_5 - t_6$
  1. *Worker* c *computes* $CTS_3 = \{y_1, y_3\}$ *using* π_2.
  2. *The main process advances to iteration* $i = 2$.
  3. *Worker* h *generates a new set of hitting sets.*

4. *Workers p_1 and p_3 complete plans for H_3 and H_5, producing π_{H3} and π_{H5}.*

- $t_6 - t_7$
  1. *Workers p_1, p_2, and p_3 start computing plans for H_6, H_7, and H_8, respectively.*
  2. *Worker c evaluates π_{H3} and determines it is not a valid plan.*
- $t_7 - t_8$
  1. *Worker p_2 computes $\pi_{H7} = \text{GO\_W}, \text{GO\_E}, \text{GO\_S}$.*
  2. *Since π_{H7} is the first plan found in this iteration, it becomes the candidate plan, updating $\pi_3 = \text{GO\_W}, \text{GO\_E}, \text{GO\_S}$.*
  3. *Worker c evaluates π_{H5} and determines it is not a valid plan.*
- $t_8 - t_9$
  1. *Worker c computes $CTS_4 = \{y_1, y_2\}$ using π_3.*
  2. *The main process advances to iteration $i = 3$.*
  3. *Worker h generates new hitting sets.*
  4. *Worker p_1 completes plan computation for H_6, producing π_{H6}.*
- $t_9 - t_{10}$
  1. *Worker p_1 begins computing a plan for H_9, and p_2 begins computing a plan for H_{10}.*
  2. *Worker c evaluates π_{H6} and determines it is a valid plan. **parallel-CPCES** stops all the workers and the algorithm terminates.*

*You will notice that, although **parallel-CPCES** went through a total of four iterations and identified four candidate plans, the valid plan was ultimately discovered during the third iteration. This valid plan was not one of the candidate plans selected in each iteration but was instead found among the alternative plans.*

From Figure 5.1, we can observe that since multiple plans are computed in each iteration and the first discovered plan is selected as the candidate plan, the entire process forms a tree-like structure. To describe our approach, the concept of search

depth is useful, with the classical abstraction being processed by a classical planner given a depth d in case d classical plans that are not solutions to the concrete PCP are the basis of that abstraction.

In the following subsections, we will introduce each component of `parallel-CPCES` individually and conclude by assembling them into a complete system.

5.2 Managing Counter-Tags in Parallel

The calculation of counter-tags in `p-CPCES-hit` (Line 11 of Algorithm 8) happens iteratively, processing one tag at a time. That sequential approach can lead to bottlenecks when a large number of tags need to be verified. For example, in the problem `BOMB` p100-1, where there are 100 bombs and one toilet, 100 tags must be processed. If each tag takes 0.5 seconds to compute, the entire process would take 50 seconds if done sequentially. Fortunately, since the tags are independent of one another, this task can be parallelized. In this subsection, we explain how the new parallel algorithm in `parallel-CPCES` efficiently leverages this parallelism to improve performance.

In `parallel-CPCES`, two dedicated monitors manage the computation of counter-tags. The Forward Counter-Tag Monitor (FCTM) is responsible for computing the first set of counter-tags for each depth, ensuring that initial progress moves smoothly. The Incumbent Counter-Tag Monitor (ICTM) handles all subsequent counter-tag computations throughout each depth, ensuring that remaining tasks are completed efficiently.

Both of FCTM and ICTM take a list of all tags L, PCP problem P and number of workers n as input, and both of them operate in a continuous loop, repeatedly executing the tasks without interruption. The only difference between FCTM and ICTM is that FCTM computes the CTS for the first candidate plan at each depth, while ICTM computes CTS for all the other candidate plans at any depth. As it is therefore only necessary to explain one, FCTM.

At the start of each iteration, FCTM initializes an empty list A to track workers, an empty shared queue Q for handling counter-tags, and an empty set CTS to store verified counter-tags. The shared queue Q allows multiple workers to access

Algorithm 9 [Forward | Incumbent]-Counter-Tag-Monitor (FCTM | ICTM)

Input:
- a list of all tags L,
- PCP problem P, and
- number of workers n

```
1: loop                                              ▷ Loop until system shutdown
2:     A := []                                       ▷ a list of asynchronous processes
3:     Q := []                                       ▷ initially empty shared queue
4:     CTS := ∅                                      ▷ an initially empty set of counter-tags
5:     if Forward then
6:         await_file_system_notify_earliest_plan()  ▷ notify indicates the earliest
   candidate plan has been in the database.
7:         π, d, B := get_earliest_plan_and_depth_B_from_database()
8:     else                                          ▷ Prohibit getting earliest plan at depth.
9:         await_file_system_notify_any_plan()       ▷ notify indicates the any other
   candidate plan has been in the database.
10:        π, d, B := get_any_plan_at_any_depth_and_B_from_database()
11:    L' := shuffle(L)                              ▷ Randomly reorder the element in the list
12:    SL := split_list(L', n) ▷ SL is the list of contiguous n-length sublists of L
13:    for T in SL do
14:        p = apply_async(verify_counter_tags, (SL, P, π, Q))   ▷ asynchronous
15:        A.append(p)
16:    loop
17:        F := all_workers_terminated(A) ▷ check if all workers have terminated
18:        if F := True and Q.size() = 0 then ▷ not enough counter-tags, π is a
   valid plan
19:            store_to_database(π)
20:            found_PCP_plan_signal_to_file_system(π)
21:            break
22:        if Q.size() > 0 then
23:            t := Q.pop()
24:            CTS.add(t)
25:            p := compute_probability(CTS, P)          ▷ see Section 4.4
26:            if p >= 1 − τ then                        ▷ find a CTS
27:                B := B ∪ {CTS}
28:                write_to_database(π, d, B)
29:                notify_file_system(B)
30:                break
```

Algorithm 10 verify_counter_tags

Input:
- a list of tags SL,
- PCP problem P,
- candidate plan π, and
- shared queue Q

```
1: for t in SL do
2:     if verified_counter_tag(t, P, π) then          ▷ see Section 4.3
3:         Q.push(t)
```

and coordinate efficiently. The process then waits for a notification from the file system (Line 6 in Algorithm 9), indicating that the first candidate plan for the depth d has been identified. Once the notification is received, FCTM retrieves the candidate plan π, the depth d of π, and the set of CTS B at d from the database. It is important to note that multiple candidate plans may be identified almost simultaneously. In such cases, FCTM selects the candidate plan with the highest priority, determined by the priority of the hitting set (see Section 5.3) used to generate the plan. Similarly, ICTM also chooses the candidate plan with the highest priority. Choosing a candidate plan with the highest priority ensures that both ICTM and FCTM are able to find a valid plan as early depth as possible.

Subsequently, all tags in L are shuffled to randomize their order (Line 11 in Algorithm 9) and evenly distributed among the workers (Line 14 in Algorithm 9). Each worker, stored in A, then checks its assigned tags to determine if any qualify as counter-tags (Algorithm 10). When a worker identifies a counter-tag, it adds the tag to the shared queue Q (Line 3 in Algorithm 10). FCTM continuously monitors the shared queue. When a tag becomes available in Q, it will be removed from Q and added into the set CTS (Line 24 in Algorithm 9). FCTM then calculates the probability of the counter-tags in CTS (Line 25 in Algorithm 9). This process continues until enough counter-tags have been identified to meet the probability threshold (Line 26 in Algorithm 9). Then FCTM adds the corresponding CTS into B, notifies the file system of the updated B, and stores π, depth d, and the updated B into the database. If all processing has finished and there are no more counter-tags queued in Q, it means there are not enough counter-tags for π. In this case, π is deemed valid, and the program sends signal to file system, indicat-

ing a valid plan was found (Line 29). A graphical presentation of the monitors interacting with the database and file system is given in Figure 5.2.

Instead of processing all tags from Q in one batch, we process them incrementally to form the minimal counter-tag set. While this approach might seem slower, it offers a significant advantage when dealing with unsolvable PCP problems, since with the minimal size of CTS, fewer depths are required to test hitting sets (as seen in lines 7 to 10 in Algorithm 8).

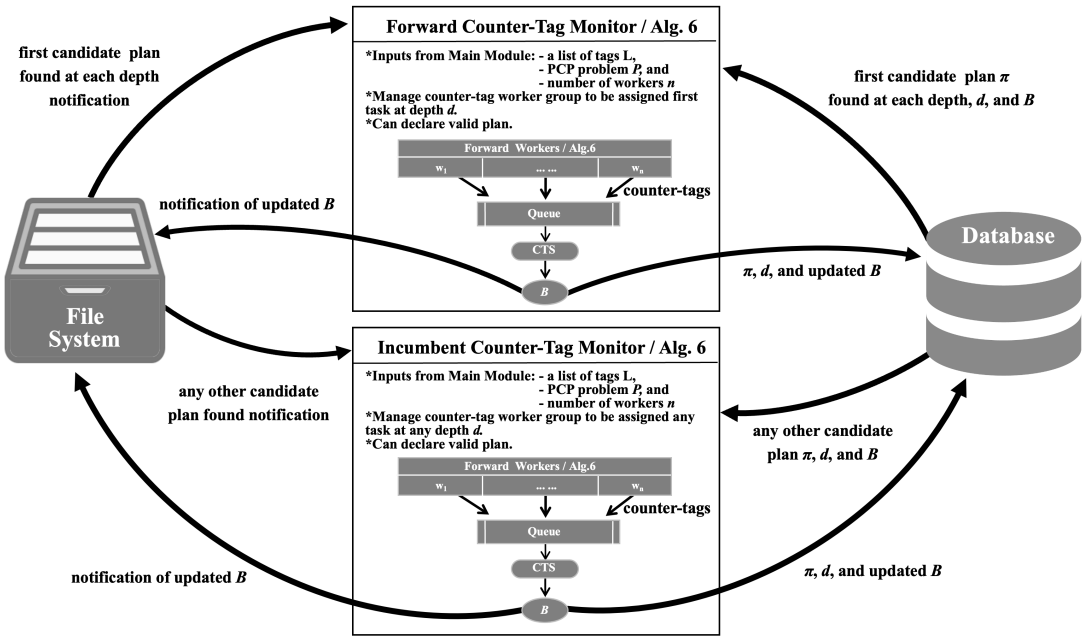


Figure 5.2: Structure of FCTM and ICTM.

Example 24. *Figure 5.3 illustrates the process of computing CTS with FCTM. For our running example, the probability threshold τ is set to 0.75, meaning the total probability of counter-tags in CTS must reach at least 0.25 for plan reach to continue. This example demonstrates the computation of CTS in the first depth, where the candidate plan is ε, depth $d = 1$ and $B = \emptyset$. The procedure of next depths is similar. We assume three workers are available for this task.*

The list of all tags L is computed by Main Module and it is the input for FCTM (Line 7 in Algorithm 15). Here, L consists of 6 tags: $L = [x_1, x_2, x_3, y_1, y_2, y_3]$.

FCTM shuffles L, producing a new random order $L' = [y_2, y_3, x_1, y_1, x_3, x_2]$, and

distribute the tags evenly among the three workers:

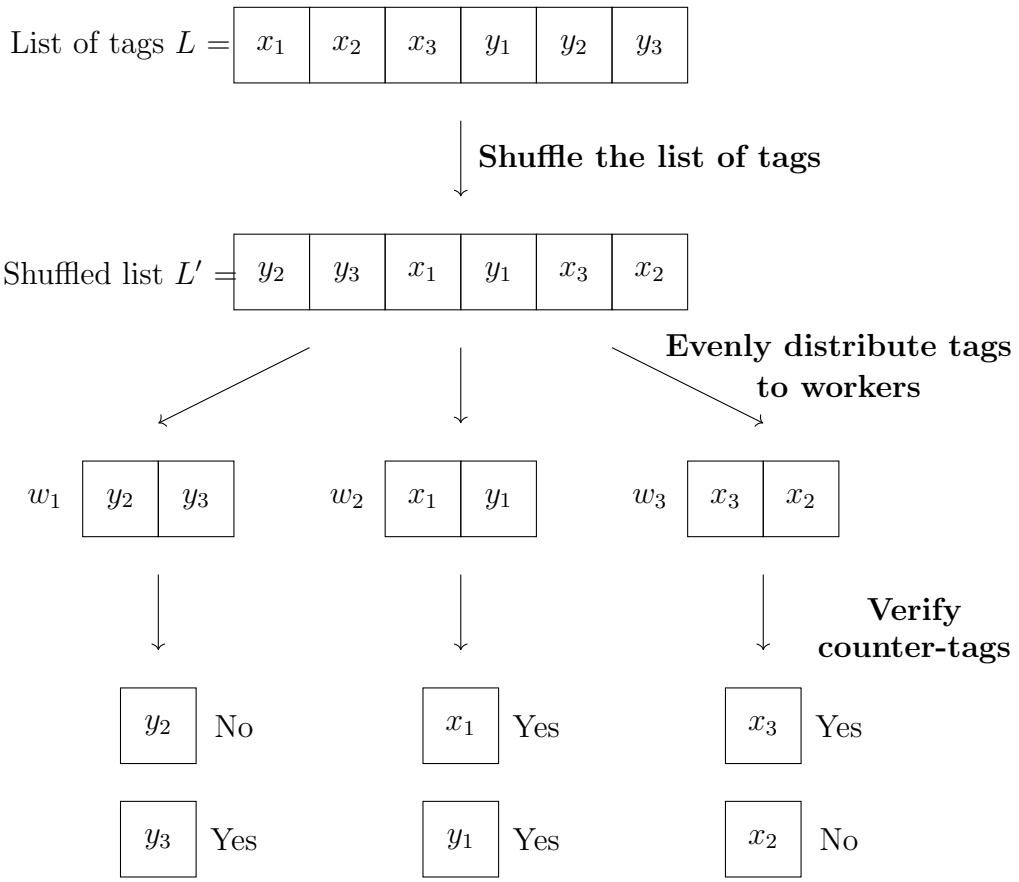
- w_1 *verifies* y_2 *and* y_3
- w_2 *verifies* x_1 *and* y_1
- w_3 *verifies* x_3 *and* x_2

Note that L can be reused in future depths, but the order of tags must be reshuffled every time it is used.

The three workers (w_1, w_2, w_3) begin verifying their assigned tags simultaneously, processing them one at a time. For instance, w_1 verifies y_2 before y_3.

If a tag is recognized as a counter-tag for the current candidate plan, it is added to the counter-tag queue Q. FCTM monitors the queue to check if there are counter-tags available.

- *The first counter-tag, x_1, is identified by w_2, and w_2 adds x_1 into Q.*
- *At this point, w_1 and w_3 have not yet finished verifying their first tags.*
- *FCTM detects one counter-tag in Q and removes x_1, adding it to CTS. FCTM calculates the probability of the counter-tags in CTS, which is 0.2. Since this is less than 0.25, CTS is incomplete.*
- *While FCTM calculates the probability, w_3 finishes verifying x_3 and identifies it as a counter-tag, adding it to Q. Simultaneously, w_1 finishes checking its two tags and determines that y_3 is also a counter-tag, which is then added to Q. At this point, two counter-tags, x_3 and y_3, are in Q.*
- *Immediately after finishing the probability calculation for CTS, FCTM notifies the two counter-tags in Q. FCTM removes the first counter-tag, x_3, and adds it to CTS. Now, CTS contains x_1 and x_3. FCTM then calculates the probability of CTS, which is 0.3. Since 0.3 is greater than 0.25, CTS has reached the required probability, and the updated B is determined $B = \emptyset \cup CTS = \{\{x_1, x_3\}\}$.*
- *Finally, FCTM notifies file system the updated B, and stores $\pi = \varepsilon$, $d = 1$, and $B = \{\{x_1, x_3\}\}$ into database.*



Compute probability by FCTM—ICTM

Counter-tag queue Q = x_1 CTS = x_1 Probability = 0.2

Counter-tag queue Q = x_3 y_3 CTS = x_1 x_3 Probability = 0.3

Figure 5.3: Procedure for computing the set of counter-tags concurrently.

5.3 Managing Hitting Sets in Parallel

In Algorithm 8, the inner loop continuously computes a hitting set that has never appeared before, until either a candidate plan is found or none of the hitting sets can give us a candidate plan. This means that each candidate plan per iteration is derived from the first solvable problem associated with the hitting set found. However, for many PCP problems there are multiple solvable problems associated with hitting sets in each iteration. The choice of which hitting set to use directly impacts the efficiency of `p-CPCES-hit`. For example, we observed that if `p-CPCES-hit` selects the smallest hitting set, its efficiency can be lower than when a hitting set is selected at random (see Section 4.7.2).

`parallel-CPCES` features multiple workers available to compute hitting sets and candidate plans. This presents an opportunity to select multiple distinct hitting sets for candidate plan generation. Unlike Algorithm 8, which produces only one minimal or random hitting set per iteration, our strategy generates multiple hitting sets each depth and assigns each one a priority level. We then select the highest-priority hitting sets in order to pose problems for which we will compute a candidate plan. Since multiple workers are calculating plans at each depth, we designate the first plan found as the candidate plan for that depth.

This approach has two key advantages. First, `parallel-CPCES` selects only the good hitting sets (with highest priority), which significantly enhances search efficiency. Second, by using the first plan found as the candidate plan for each depth, `parallel-CPCES` saves planning time and quickly progresses to the next depth. This contrasts with `p-CPCES-hit`, where only one hitting set is used per round where either an unsolvable hitting set is encountered, requiring recalculation with the next one, or the planning time for each round is entirely dependent on the time required for the current hitting set.

In Example 23 multiple hitting sets are computed from a set of CTS at each depth. However, we observed that simply generating hitting sets in this manner does not substantially improve efficiency. To expedite finding a valid plan, we use a special method for generating hitting sets, which includes randomly generated seed hitting sets as well as derivative hitting sets derived from these seed hitting sets.

Definition 19. *A seed hitting set SH is a hitting set of the counter-tag sets B.*

In other words, in `parallel-CPCES`, a seed hitting set is the hitting set generated by function `unique_hit` in Algorithm 8.

Definition 20. *Given a list of tags L from a PCP problem, a derivative hitting set DH(SH) is a set derived from a seed hitting set SH such that DH is the union of SH with any number of new tags from L: DH(SH) = SH ∪ l where l ⊆ (L\SH).*

These derivative hitting sets are created by adding new tags to the seed set, expanding the diversity of potential candidate plans. The tags used to form derivative hitting sets do not have to be counter-tags; any tags can be included to increase diversity.

Theorem 2. *A candidate plan that satisfies a derived hitting set must also satisfy the corresponding seed hitting set.*

Proof of Theorem 2. As mentioned in Section 4.6, given a hitting set $H = \{h_1 \cdots h_n\}$ of the counter-tag sets B, a candidate plan $\Pi(P_H)$ will belong to the set $\Pi(P_H) = \Pi(P_{1h_1}) \cap \cdots \cap \Pi(P_{nh_n})$. After adding an arbitrary number of new tags $\{t_1, \cdots t_n\}$ into H, the derived hitting set becomes $H' = \{h_1, \cdots, h_n, t_1, \cdots, t_n\}$. Each new tag t_i can be regarded as part of a classical problem P_{ti}, where the goal condition is the tag t_i itself. Therefore, a candidate plan $\Pi(P_{H'})$ for H' will belong to the set $\Pi(P_{H'}) = \Pi(P_{1h_1}) \cap \cdots \cap \Pi(P_{nh_n}) \cap \Pi(P_{t1}) \cap \cdots \cap \Pi(P_{tn})$. Since $\Pi(P'_H) \subseteq \Pi(P_H)$, $\Pi(P_{H'})$, any plan that satisfies the derived hitting set H' must also be a valid candidate plan for the original hitting set H. □

Definition 21. *The priority of a hitting set equals the number of tags in this hitting set*

Given the limited number of workers available to compute candidate plans, we assign priorities to hitting sets to ensure efficient use of resources. We choose to use the number of tags in the hitting set as the priority of such hitting set because the more tags it contains, the higher likelihood set of states from the initial state distribution is. Although calculating the exact probability of initial states would be ideal, it is computationally expensive, so we use tag count as a pragmatic alternative.

Computing hitting sets is a relatively simple task that can be accomplished using a busy loop, allowing hundreds of hitting sets to be generated within a second. Due to this efficiency, a single worker is sufficient to generate an adequate number of hitting sets at each depth. In `parallel-CPCES`, we employ a Hitting Set Monitor (HSM) to manage the workers responsible for computing hitting sets. Each worker is dedicated to computing hitting sets for a specific depth. When the main process advances to the next depth, if a worker from a previous depth has not yet finished computing all hitting sets, HSM does not terminate its execution. Instead, it allows the worker to continue generating hitting sets. For example, suppose there are two workers, h_1 and h_2, managed by HSM. When the main process reaches $depth = 1$, h_1 computes the hitting sets for this depth. Upon advancing to $depth = 2$, h_2 takes over and begins computing hitting sets for $depth = 2$. If h_1 has not yet completed its task—particularly in cases where the number of hitting sets is extremely large and exhaustive enumeration is time-consuming—`parallel-CPCES` allows h_1 to continue running until the main process reaches $depth = 3$. At this point, both h_1 and h_2 may still be computing their respective hitting sets. However, since no additional workers are available for $depth = 3$, `parallel-CPCES` instructs HSM to terminate h_1 and reassign it to compute hitting sets for $depth = 3$. This approach is motivated by the observation that when a large number of hitting sets exist at a given depth, they cannot be computed instantaneously. `parallel-CPCES` requires these hitting sets to compute their priorities and select those with the highest priority for plan computation. In practice, when a worker responsible for computing plans becomes available, `parallel-CPCES` queries the database to check for hitting sets ready for plan computation. This querying process takes time, and within this period, the hitting set workers typically generate only dozens of hitting sets. Consequently, `parallel-CPCES` must select the highest-priority hitting sets from those already available in the database. As `parallel-CPCES` moves to deeper depths, the number of hitting sets generated at each depth decreases, eventually falling below the number of plan computation workers. At this stage, the surplus planning workers are reassigned to compute plans for previous depths. This provides an opportunity for `parallel-CPCES` to access high-priority hitting sets, facilitating the rapid discovery of a valid plan. This phenomenon not only occurs when the main process advances to deeper depths but also when computing CTS becomes a

Algorithm 11 Hitting-Set-Worker

```
    Input:
     - a list of all tags L, and
     - current depth d
 1: store_to_database("status_running", d)
 2: B := get_from_database_B_of_first_plan_at(d)
 3: loop
 4:     SH := compute_seed_hitting_set(B)          ▷ this is a random hitting set
 5:     if SH = ∅ then
 6:         store_to_database("status_finished", d)
 7:         process terminates
 8:     priority := compute_priority(SH)
 9:     store_to_database(SH, d, priority)
10:     notify_file_system_new_hitting_set(SH)
11:     loop
12:         if termination_signal() then           ▷ termination signal from HSM
13:             store_to_database("status_killed", d)
14:             process terminates
15:         DH := generate_derivative_hitting_set(SH, L)
16:         if DH = None then
17:             break
18:         priority := compute_priority(DH)
19:         store_to_database(DH, d, priority)
20:         notify_file_system_new_hitting_set(DH)
```

bottleneck. If the CTS computation time is prolonged, plan computation workers may also be reassigned to compute plans from previous depths while waiting for the CTS, thereby enhancing overall efficiency.

The workers in HSM use two key strategies to prioritize derivative hitting sets:

- It prioritizes derivative hitting sets from the deepest depth over older ones.
- It processes hitting sets within the same depth based on their priority.

This prioritization directs the parallel system progress by focusing resources on the current maximum depth. By assigning more workers to compute candidate plans for the current depth, the system is more likely to find a candidate plan quickly, enabling it to move to a greater depth sooner.

Algorithm 11 details the process of generating hitting sets by a worker managed by HSM. The algorithm starts by updating the database to indicate that the worker is "running" for depth d and retrieves B from the database at encountered depth d. It then enters an outer loop to generate random seed hitting sets (Line 15). For each seed hitting set generated, it is recorded in the database for other components to access. If no seed hitting sets can be generated, it indicates that all potential seed hitting sets have been explored, so the algorithm updates the database with a "finished" status and terminates (Line 5). Once a seed hitting set is available, the algorithm computes the priority to this seed hitting set, stores information into database and notifies the file system. Next, the algorithm enters an inner loop to generate derivative hitting sets by adding arbitrary tags to the seed set (Line 15). The priority of each derivative hitting set is computed, and the worker stores information for derivative hitting sets into database and puts a notification to the file system. This process continues until no further derivative hitting sets can be generated, at which point the inner loop ends, and the outer loop moves to the next seed hitting set. Note that in Algorithm 8, we must verify that all hitting sets are unsolvable before we can conclude that the current PCP problem has no solution. Similarly, in `parallel-CPCES`, we must confirm that all seed hitting sets are unsolvable before asserting that the current PCP problem is unsolvable. This is the reason why this algorithm moves to the next seed hitting set after inner loop (Line 11) completes, and only if all seed hitting sets and derivative hitting sets are generated, the worker can terminate. The algorithm also checks for a termination signal at the beginning of each inner loop cycle (Line 12). If the Main Module (MM, main processing) issues a termination signal, the HSM updates worker's status to "killed" in the database and stops its execution. This termination mechanism allows for orderly shutdowns when the problem has been solved.

Figure 5.4 illustrates the structure of HSM. Here, two workers are allocated to HSM to manage hitting set computations. When an updated B becomes available for a specific depth d, HSM activates workers assigned at that depth d. The worker retrieves B at depth d from the database, which is needed for the hitting set calculations. A worker dedicated to the current depth d begins by generating

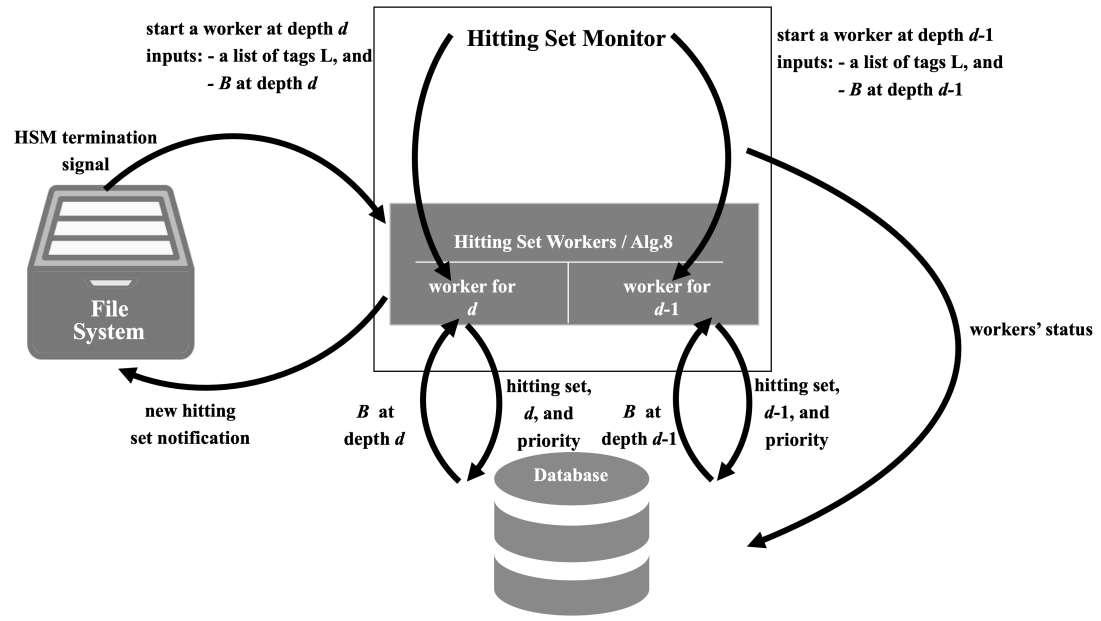


Figure 5.4: Structure of HSM.

both seed and derivative hitting sets. Each generated hitting set is recorded in the file system and stored in the database. Notifications of new hitting sets are sent to the file system to inform other modules. Meanwhile, hitting sets for the previous depth $(d-1)$ continue to be processed concurrently. This enables continuous and efficient hitting set generation across different depths, maximizing the use of computational resources and minimizing delays in the search process.

5.4 Managing Candidate Plans in Parallel

In Algorithm 8, only a single worker is assigned to compute candidate plans, which makes the search linear. Problems are posed in turn for each hitting set, with the final encountered problem in this sequence either yielding a concrete plan, or otherwise allowing the conclusion that none exists. In contrast, in the approach discussed in Section 5.3, multiple hitting sets are generated at each encountered depth, and multiple workers can explore a range of problems at each depth.

Our parallel approach incorporates a Candidate Plan Monitor (CPM) which manages a pool of classical planning processes. Problems posed given hitting sets

are planned for independently in parallel, and we develop a prioritisation scheme for getting the most out of planning systems. By having multiple workers, `parallel-CPCES` can explore multiple problems at each depth, thereby increasing the chances of finding a valid plan relatively quickly.

Before we present the formal detail, to make the intuitions here concrete in Figure 5.6 we have a schematic view that exhibits a scenario in which multiple workers are operating in parallel. Here, workers $\{w_1 \cdots w_n\}$ are assigned to compute candidate plans. At every encountered depth there is always a *idle worker* — i.e., the worker who first finished the task for last depth — responsible solely for computing candidate plans for the first seed hitting sets of next depth. The remaining workers are assigned to compute candidate plans related to derivative hitting sets and all other seed hitting sets. Our approach ensures that an idle worker is always available to compute candidate plans. In each depth, as soon as a seed hitting set SH_1 is computed, it is assigned to an idle worker (w_1 for example). Our system features parallel processes that continuously generate derivative hitting sets from SH_1: $DH_1(SH_1)$, $\cdots$, $DH_n(SH_1)$, calculating the priority of each. In priority order planners $\{w_2 \cdots w_n\}$ are assigned to work associated with derivative hitting sets. It is important to note that workers do not wait for all derivative hitting sets to be generated before starting to compute candidate plans. Instead, they begin as soon as some derivative hitting sets are available. In each depth, our approach chooses the first plan found as the candidate plan to stimulate the main processing entering into next depth. Any additional plans discovered at that depth are only verified by ICTM to check whether a solution to the concrete PCP at hand. If SH_1 and all its derivative hitting sets $DH_1(SH_1)$, $\cdots$, $DH_n(SH_1)$ are unsolvable, the second seed hitting set SH_2 and its derivative hitting sets $DH_1(SH_2)$, $\cdots$, $DH_n(SH_2)$ will be computed and used. If no solution is found from any of the seed hitting sets or their derivative hitting sets, then the PCP problem has no solution.

Example 25. *We explain this process by using our running example. Assume that four workers $\{p_1, p_2, p_3, p_4\}$ responsible for computing candidate plans in our parallel system. Unlike Example 23, we now assume that computing three hitting sets requires 1 second, reflecting the reality that obtaining hitting sets incurs computa-*

tional cost. The entire planning process takes 10 seconds, denoted by $\{t_0, \cdots, t_{10}\}$. *The processing is presented in Figure 5.5, in which black dots represent the time taken to compute a plan from a hitting set and red dots represent the time taken to compute CTS for the corresponding plan. For brevity, at each depth, we display at most four hitting sets in the figure. Additional hitting sets generated are omitted and are not used in plan computation.*

- t_0
  1. ***parallel-CPCES*** *starts execution at depth* $d = 1$ *with an initial candidate plan* $\pi_0 = \varepsilon$.
- $t_0 - t_1$
  1. *FCTM computes* $CTS_1 = \{x_1, x_3\}$.
  2. *HSM instructs the workers to compute hitting sets for depth* $d = 1$, *and gets three hitting sets: one seed hitting set* SH_1 *and two derivative hitting sets* DH_{11}, DH_{12}.
- $t_1 - t_2$
  1. *Workers* p_1, p_2, p_3 *compute plans for* SH_1, DH_{11}, *and* DH_{12}, *respectively.* p_4 *is an idle worker waiting for the first seed hitting set of next depth.* p_1 *finds a plan* $\pi =$ GO_W *within this second.*
  2. *The main processing advances to depth* $d = 2$, *updating the candidate plan to* $\pi_1 =$ GO_W.
  3. *HSM generates three additional derivative hitting sets, but only* DH_{13} *is displayed due to space constraints.*
  4. *The Main Module instructs HSM to stop computing hitting sets at* $d = 1$.
- $t_2 - t_3$
  1. *FCTM computes* $CTS_2 = \{x_2\}$ *from* π_1.
  2. *HSM instructs workers to compute hitting sets for* $d = 2$, *yielding one seed hitting set* SH_2 *and two derivative hitting sets* DH_{21}, DH_{22}.
- $t_3 - t_4$
  1. p_4 *starts computing the plan for* SH_2, *and* p_1 *becomes the idle worker.*

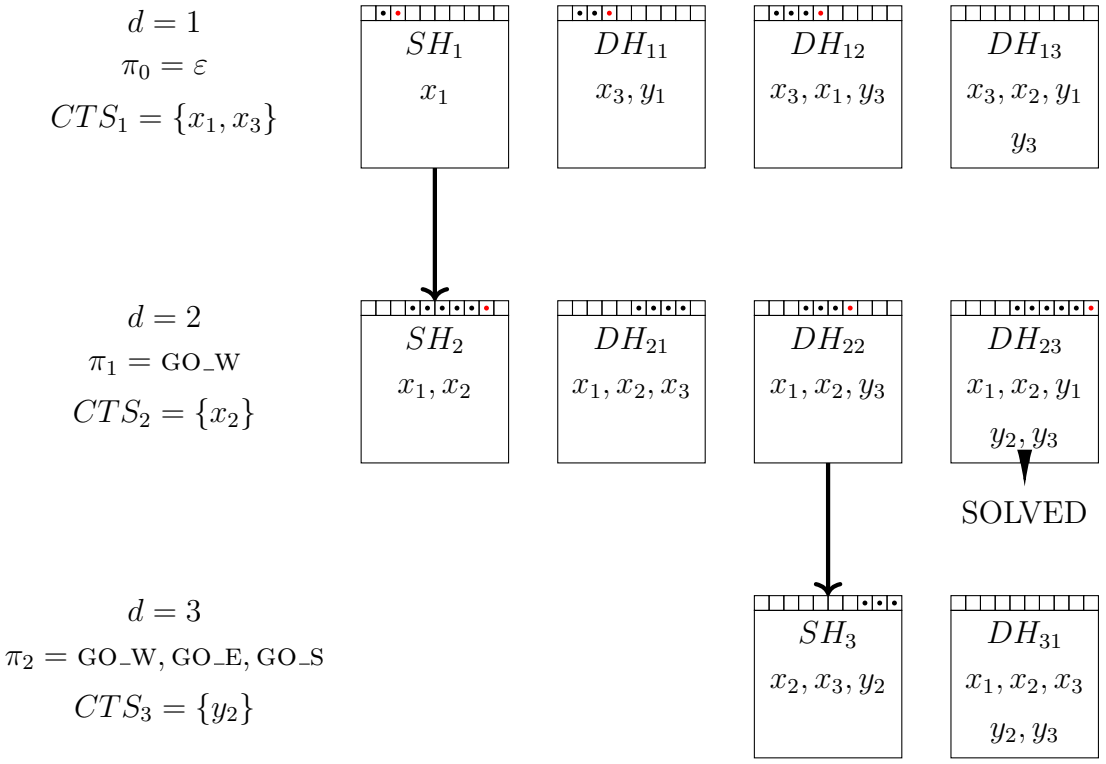


Figure 5.5: Operation of `parallel-CPCES` with derivative hitting sets and their priorities. Each large box represents a classical abstraction of the PCP problem parameterized by a hitting set. The array of cells above each box provides a linear timeline: a black token indicates that a classical planner is scheduled to solve the problem at that time, and a red token indicates a counter-tag set calculation based on the generated plan.

2. *p_2 completes the plan computation for DH_{11}, and ICTM computes its CTS.*

3. *Right now there are three hitting sets (DH_{13}, DH_{21}, DH_{22}) waiting to be used to compute plan. Since DH_{21}, DH_{22} owns the same highest priority within them, p_2 randomly selects DH_{22} for computing plan.*

4. *HSM generates three new derivative hitting sets, but only DH_{23} is displayed.*

- $t_4 - t_5$

  1. *p_3 completes the plan computation for DH_{12} and ICTM computes its CTS.*

  2. *Among the remaining hitting sets (DH_{13}, DH_{21}, DH_{23}), DH_{23} has the highest priority. So, p_3 selects DH_{23} for plan computation.*

- $t_5 - t_6$

  1. *p_2 finds a plan $\pi = \text{GO\_W}, \text{GO\_E}, \text{GO\_S}$ for DH_{22}. The main processing advances to $d = 3$, updating the candidate plan to $\pi_2 = \text{GO\_W}, \text{GO\_E}, \text{GO\_S}$.*

  2. *Among the remaining hitting sets (DH_{13}, DH_{21}), DH_{21} has the highest priority. So, p_2 selects DH_{21} for plan computation.*

  3. *The Main Module instructs HSM to stop computing hitting sets at $d = 2$.*

- $t_6 - t_7$

  1. *FCTM computes $CTS_3 = \{y_2\}$ from π_2.*

  2. *HSM instructs workers to compute hitting sets for $d = 3$, yielding one seed hitting set SH_3 and two derivative hitting sets DH_{31}, DH_{32}. Due to space limit, only DH_{31} is displayed.*

- $t_7 - t_8$

  1. *p_1 starts computing the plan for SH_3.*

  2. *p_4 successfully computes the plan for SH_2 and ICTM computes its CTS. Since p_4 is the only free worker right now, p_4 becomes an idle worker.*

Algorithm 12 Candidate-Plan-Monitor (CPM)

Input:
- a PCP problem $\mathcal{P}$, and
- number of workers for CPM n

```
1: idle_worker_depth := 1
2: pool := initialize_workers_pool(n)
3: loop
4:     await_hitting_set()          ▷ continues if unprocessed hitting set in database
5:     H, d := get_hitting_set_and_depth()
6:     if d = idle_worker_depth then
7:         apply_async(plan_search, (P, H, d))   ▷ an idle worker solves first task at d
8:         special_worker_depth++
9:     else
10:        await_free_regular_worker(pool)   ▷ continues if at least two workers are free
11:        apply_async(plan_search, (P, H, d))        ▷ assign a regular worker to problem
```

- $t_8 - t_9$

  1. *ICTM computes CTS for SH_2.*
  2. *p_3 completes the plan computation for DH_{23} and p_2 completes he plan computation for DH_{21}. At this point, ICTM is presented with two plans and must choose one for verification. Since DH_{23} has a higher priority than DH_{21}, ICTM selects DH_{23} to compute its CTS.*

- $t_9 - t_{10}$

  1. *ICTM detects that the plan generated from DH_{23} is a valid solution. Parallel system terminates.*

Algorithm 12 describes CPM process for managing candidate plan generation in parallel. Initially, CPM sets a counter called "idle_worker_depth" to 1, which is used to track the depth of the hitting set assigned to the idle worker, and initializes the pool of workers. An idle worker handles the first seed hitting set at each depth to avoid delays when progressing to the next depth. The idle worker is not a fixed worker, rather at each depth `parallel-CPCES` designates one worker to keep idle until the next depth. That worker remains idle until the algorithm transitions to

Algorithm 13 candidate_plan_searching_task

Input:
- a PCP problem P
- a hitting set H, and
- hitting set depth *depth*

1: store_to_database("status_running", d, H)
2: π := search_candidate_plan(P, H) ▷ classical planning task (Sections 4.5 - 4.6).
3: store_to_database(π, d)
4: **if** $\pi \neq \perp$ **then**
5: notify_file_system(π) ▷ task is solvable
6: store_to_database("status_finished", d, H)

the next depth. Once the transition occurs, that idle worker is assigned the first planning task of the new depth, ensuring an efficient continuation of the planning process. CPM then enters a loop where it waits for a notification from the file system indicating that a new hitting set has been stored in the database (Line 4). Once notified, CPM retrieves the hitting set with the highest priority along with its corresponding depth d. If d matches "idle_worker_depth" (indicating it is the first hitting set for that depth), CPM assigns the task to the idle worker (Line 6) and increments "idle_worker_depth" by one to prepare for the next depth. For any other hitting sets at this depth and previous depths, CPM waits until a regular worker is available (Line 10). Once at least two workers become available in the pool, CPM assigns the candidate plan computation task to one of them (Line 11). The reason for keeping at least two available workers is to ensure that one idle worker is always reserved for the next depth, allowing a seamless transition in the planning process.

Algorithm 13 describes how each worker in CPM completes its task. Each worker begins by storing their running status into database, and then attempting to find a candidate plan (the method has been introduced in Section 4.5 and 4.6). Once this search concludes, one of two outcomes is possible: either a candidate plan is identified, or no solution is found. In both cases, the result is recorded in the database to maintain a log of all outcomes. If the first plan at encountered depth is found, the worker proceeds to notify the file system to signal that the candidate

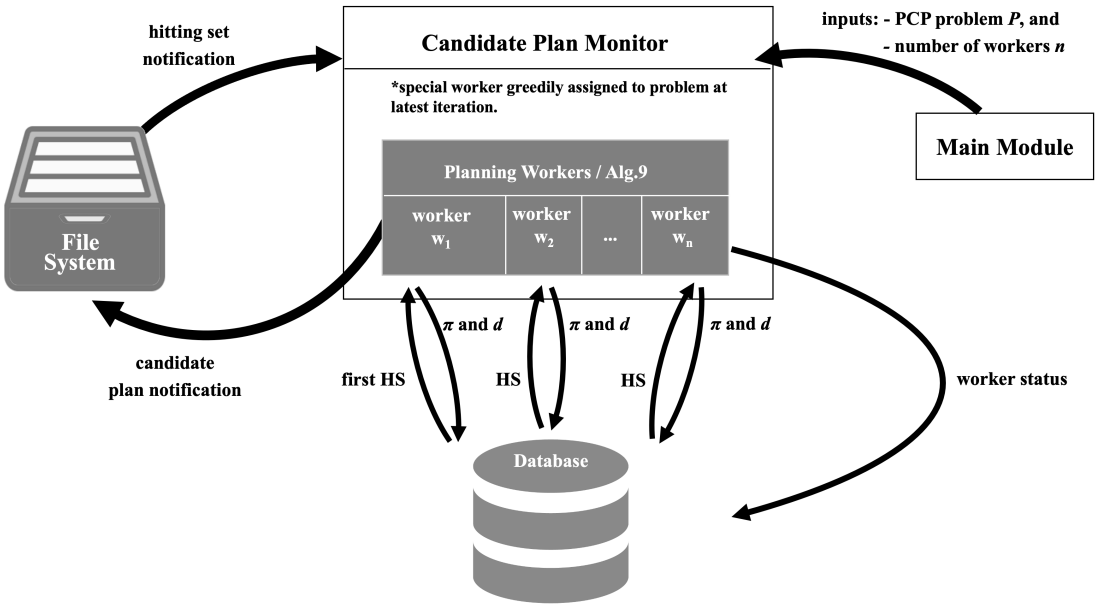


Figure 5.6: Structure of CPM. "HS" denotes hitting set.

plan has been successfully identified. At last, the worker update their status as terminated in database.

5.5 A Parallel CPCES-Based Planner for Probabilistic Conformant Planning

In this subsection, we bring together the FCTM, ICTM, HSM, and CPM to construct the complete `parallel-CPCES` system. At the core of this system is the Main Module (MM), which oversees the entire process. The MM is responsible for the following tasks:

- Managing monitors: The MM initializes and terminates all monitors, including FCTM, ICTM, HSM, and CPM.
- Tracking depths: The MM keeps track of the depth count. Each time the first plan is found at a depth, the MM increments the depth counter by one.
- Determining problem solvability: The MM includes a mechanism to manage skipping for candidate plan searching (Algorithm 14), which continuously

checks both the file system and database to check whether the PCP problem is known to be unsolvable. If no solution exists, Algorithm 14 notifies the MM.

- Managing skipping for candidate plan searching: To optimize the search process, the MM queries the database to check if a hitting set from the encountered depth was already encountered in a previous depth. If the same hitting set was processed in an earlier depth and its candidate plan has already been found, the MM skips further exploration of the encountered depth to avoid redundant computations.

5.5.1 Managing Skipping for Candidate Plan Searching

`parallel-CPCES` avoids computing multiple candidate plans for the same hitting set, as that set occurs at different depths, by re-using rather than re-computing plans. In this case, we say it skips the computation of a plan. Because hitting sets are frequently repeated between depths, our mechanism for skipping plan re-computation and using a lookup mechanism to retrieve previously computed plans accelerates the overall search.

Example 26. *The following example illustrates how MM performs depth skipping.*

- *Depth 1: The counter-tags to $\pi_0 = \varepsilon$ identified are x_1 and x_3. A seed hitting set $SH_1 = \{x_3\}$ is generated, along with three derivative hitting sets: $DH_{11} = \{x_3, y_1\}$, $DH_{12} = \{x_3, x_2\}$, and $DH_{13} = \{x_3, x_2, y_1, y_3\}$. The candidate plan $\pi_1 = \text{GO\_W}$ is found for SH_1.*
- *Depth 2: The counter-tags for π_1 are x_1 and x_2. During the candidate plan search, the plan for DH_{12} found is $\pi'_1 = \text{GO\_W}, \text{GO\_E}$. Meanwhile, the worker managed by HSM generates a new seed hitting set $SH_2 = \{x_2, x_3\}$, which is the same as the earlier hitting set DH_{12}. Since the candidate plan for DH_{12} was already computed, MM skips further plan searches and directly proceeds to depth 3 with $\pi_2 = \pi'_1$.*

Algorithm 14 is designed to optimize the process of searching for a candidate plan by checking if any plan from previous depths can be reused, thus potentially

Algorithm 14 skip_plan_search

```
1: Input: depth, d
2: Output: a plan π_skip or ⊥
3: HS := get_hitting_sets_from_database(d)      ▷ a list of hitting sets at depth d
4: for h in HS do
5:     π_skip = get_plan_for(h, d)              ▷ π_skip is due to h at some depth
   [d − i|d ≥ i > 0]
6:     if π_skip ≠ ⊥ then
7:         return π_skip
8: return ⊥
```

skipping redundant computations. The algorithm begins by retrieving all hitting sets (HS) generated for depth d from the database (Line 3). For each hitting set in HS, it checks if there is a matching hitting set from a previous depth that already has an associated candidate plan (Line 5). If a matching plan is found, it immediately returns that plan (Line 7), allowing the system to skip the current candidate plan search. If no matches are found, the algorithm returns $\perp$ (Line 8).

Note that Algorithm 14 is invoked repeatedly in the loop starting Line 17 in Algorithm 15. Between invocations the HSM may have produced additional hitting sets, and therefore the population of hitting sets that might be solved by an existing plan can increase between invocations.

5.5.2 Main Module

Algorithm 15 is the Main Module responsible for managing all the monitors and processes in the `parallel-CPCES` system. An explanation of its operation, broken down step-by-step, is as follows:

1. **Initialization:** MM initializes by setting depth d to 1, initializing an empty set of counter-tag sets, B, and assigning candidate plan π as ε. It stores that initial information in the database and notifies the file system that a candidate plan is available (Line 2 to 6).

2. **Compute Tags and Start Monitors:** MM then computes all tags for the given PCP problem P and saves them in a list L. After this, it initiates all

Algorithm 15 Main Module (MM)

Input:
- a PCP problem P,
- number of workers for FCTM fn,
- number of workers for ICTM in,
- number of workers for HSM hn, and
- number of workers for CPM pn

1: **Output:** a plan π for P, or UNSAT
2: $d := 1$
3: $B := \emptyset$
4: $\pi := \varepsilon$
5: store_to_database(d, B, π)
6: notify_file_system_π()
7: L := compute_all_tags(P) ▷ a list of all tags for P
8: apply_async(Forward-Counter-Tag-Monitor, (L, P, fn))
9: apply_async(Incumbent-Counter-Tag-Monitor, (L, P, in))
10: apply_async(Candidate-Plan-Monitor, (P, pn))
11: apply_async(Hitting-Set-Monitor, (L, d, hn))
12: **loop**
13: await_counter_tag_sets() ▷ Blocking until a counter tag set at depth d is added to B
14: **if** d>hn **then**
15: free_an_HSM_worker_at_depth($d - hn$) ▷ free an HSM worker at $d - hn$
16: start_an_HSM_worker(d) ▷ start an HSM at d
17: **loop**
18: π_{skip} := skip_plan_search(d) ▷ see Algorithm
19: **if** $\pi_{skip} \neq \perp$ **then** ▷ can use previous plan
20: $\pi := \pi_{skip}$
21: d++
22: **break**
23: π, plan_type := get_earliest_plan_status(d) ▷ see Algorithm 16
24: **if** plan_type = "unsolvable" **then** ▷ P has no plan
25: kill_all_processing()
26: **return UNSAT**
27: **else if** plan_type = "valid" **then** ▷ find a valid plan for P
28: kill_all_processing()
29: **return** π
30: **else if** plan_type = "continue" **then** ▷ no candidate plan at depth d yet
31: **continue**
32: **else** ▷ found a candidate plan for current depth
33: d++
34: **break**

monitors: FCTM, ICTM, CPM and HSM (Line 7 to 11).

3. **Loop for Depth Management:** MM enters a loop where it monitors and controls the depth of the search process. It waits for the counter-tag sets B at depth d to be updated by FCTM or ICTM (Line 13) and, upon notification, triggers HSM to start a worker computing hitting sets at the current depth d (Line 16). Before requesting work from HSM, MM has a mechanism to prevent delays when the number of available workers is limited. If depth d exceeds the number of workers hn, MM terminates the worker handling the shallowest depth in HSM to free up resources for depth d (Line 15).
4. **Inner Loop for Monitoring Plan Searching:** MM then enters an inner loop. It first seeks to re-use an existing plan if possible (Line 18). If a previously generated candidate plan π_{skip} is found, MM updates the current candidate plan $\pi = \pi_{skip}$, increases d, and exits the inner loop (Line 19 to 22). Otherwise, MM executes `get_earliest_plan_status` function (see Algorithm 16), which continuously checks the progress of candidate plan searches, and returns a tuple. The first element is a candidate plan and the second element is a string indicating the current status of the search (Line 23). Based on that returned tuple:
   - If all hitting sets at the encountered depth have been generated and none of them has a candidate plan, MM concludes that the PCP problem is unsolvable and returns UNSAT (Line 24 to 26).
   - If a valid plan is found, MM receives this notification from file system (sent from Algorithm 9 Line 20), terminates all processes, and returns the valid plan (Line 27 to 29).
   - If the search is still ongoing, MM re-enters the inner loop to continue monitoring for plans (Line 30 to 30). At every iteration of the inner loop, there is a brief delay of a few seconds as the `get_earliest_plan_status` function attempts to find a result. During this time, additional hitting sets are likely to have been generated. In the next (and future) iteration of the inner loop, MM re-checks whether any of these new hitting sets are duplicates of those generated in previous depths and if a candidate plan for one of them has already been found. This mechanism allows

MM to skip some candidate plan searches, optimizing the process and reducing redundant plan searches.

- If a candidate plan has been found, MM advances to the next depth (Lines 32 to 34).

Algorithm 16 get_earliest_plan_status

```
1: Input: current depth, d
2: Output: a tuple with first element π or ⊥, and second element is a status string
3: F := check_for_plan()                ▷ nonblocking query; F = True iff PCP solved
4: if F = True then
5:     π := get_valid_plan_from_database()
6:     return π, "valid"
7: π := get_plan_from_database(d)       ▷ nonblocking query; returns a candidate plan π or ⊥
8: if π ≠ ⊥ then
9:     return π, "candidate plan found"
10: else
11:     if HSM_finished_at(d) and all_problems_processed_at(d) then
12:         return ⊥, "unsolvable"
13:     else
14:         return ⊥, "continue"
```

In Algorithm 15, the function `kill_all_processing` (used in Lines 25 and 28) terminates all monitors except for the MM, along with any classical planners working on searching for candidate plans, and workers actively computing hitting sets. This mechanism ensures MM remains active, enabling it to return results as intended (Lines 26 and 27).

Algorithm 16 describes the progress of checking the status of plan searching activities. It starts by checking the file system to determine if a valid plan has already been found by either the FCTM or ICTM (Line 3). If a valid plan π is found, the monitor returns (π, "valid"). If no valid plan is found, the monitor queries the database to check for a candidate plan at the encountered depth (Line 7). If a candidate plan π exists, it returns (π, "candidate plan found"). If no candidate plan is found, the monitor further checks if all possible work assigned to the HSM pool at the encountered depth is completed (Line 11). If all have completed their

tasks and no valid plan has been identified, the monitor returns ($\perp$, "unsolvable"), indicating that there is no solution at the encountered depth. If the planning processes are not yet complete, the monitor returns ($\perp$, "continue"), signaling that the inner loop of MM should continue monitoring planning results.

Algorithm 17 Computing the Probability of Initial States.

$$val(N) = \begin{cases} Pr(p), & \text{if } N \text{ is a leaf node } p \text{ where } p \text{ is a chance variable;} \\ 1 - Pr(p), & \text{if } N \text{ is a leaf node } \neg p \text{ where } p \text{ is a chance variable;} \\ \prod_i val(N_i), & \text{if } N = \bigwedge_i N_i; \\ \sum_i val(N_i), & \text{if } N = \bigvee_i N_i. \end{cases}$$

5.6 Experimental Results

In our evaluation, we tested four distinct probability thresholds: $\tau \in \{0.99, 0.90, 0.75, 0.5\}$. We used `FF` (Hoffmann 2001) to solve the posed classical planning problems. Additionally, we tested four different configurations for the number of workers allocated to CPM: 32, 16, 8, and 1. The benchmarks used in this experiment match those described in Section 4.7. We ran each instance nine times to reduce the effect of nondeterminism, and here we report the median results. A timeout of 1800 seconds was set for each run. All experiments reported in this section were conducted on a computing system equipped with an AMD Ryzen Threadripper 3990X 64-Core Processor and 128GB of RAM.

From our previous experiments we observed that assigning a large number of workers to FCTM and ICTM caused database locking issues due to the resulting high frequency of read and write operations. To mitigate this problem we limited the allocation to 2 workers each for FCTM and ICTM. Moreover, we found that increasing the number of workers for the HSM beyond 2 did not yield additional performance benefits. In particular, when candidate plans take significant time to generate, increasing the number of workers for HSM beyond two does not lead to performance improvement. This is because each HSM can generate dozens of hitting sets per second, which sufficiently meets the demands of CPM. Therefore,

we opted to assign 2 workers to HSM as well.

5.6.1 Efficiency of Combining Seed Hitting Sets and Derivative Hitting Sets

In previous experiments we observed that using only seed hitting sets to characterize classical abstractions — e.g., `parallel-CPCES-seed`, as illustrated in Example 23 — is not efficient. We validate that the combined use of seed hitting sets and derivative hitting sets, via `parallel-CPCES-both`, significantly improves efficiency compared to using seed hitting sets alone, we conducted comparative experiments on seven problems: `UTS` p8 ($\tau = 0.99$), `COINS` p16 ($\tau = 0.9$), `ONEDISPOSE` p3-2 ($\tau = 0.9$), `DISPOSE` p4-3 ($\tau = 0.75$), `ONEDISPOSE` p3-2 ($\tau = 0.75$), `COINS` p20 ($\tau = 0.5$), and `DISPOSE` p4-3 ($\tau = 0.5$). We selected these instances because they can be solved in a relatively short time, allowing us to quickly verify the results.

We tested `parallel-CPCES-both` and `parallel-CPCES-seed`, both configured with 16 workers in CPM, 2 workers in FCTM, 2 workers in ICTM, and 2 workers in HSM. Each instance was tested 50 times. For each instance, we compared the longest runtime of `parallel-CPCES-both` with the shortest runtime of `parallel-CPCES-seed` across the 50 results. The results showed that the shortest runtime of `parallel-CPCES-seed` was consistently longer than the longest runtime of `parallel-CPCES-both`. This result demonstrates that the combined use of seed hitting sets and derivative hitting sets indeed significantly enhances the efficiency of `parallel-CPCES`.

5.6.2 Performance of `parallel-CPCES`

To assess how the number of workers allocated to CPM influences the performance of `parallel-CPCES`, we conducted experiments with different worker counts. Intuitively, we expect that increasing the number of CPM workers will improve performance. More workers allow for the generation of more candidate plans at each depth, potentially leading to an earlier discovery of a valid plan by FCTM and ICTM. Any such performance gain should be especially noticeable when solv-

ing problems that are challenging for the classical planner for which the candidate plan searches take a relatively long time.

Figure 5.7: Relationship between the number of workers and runtime (seconds) for DISPOSE p8-2 with a probability threshold of $\tau = 0.9$. The depths at which valid plans were found are: 17 (1 worker), 10 (8 workers), 12 (16 workers), and 8 (32 workers).

As expected, some problems achieve their best performance when 32 workers are assigned. For instance, with a probability threshold $\tau = 0.9$, the DISPOSE p4-3 problem (Figure 5.10) takes 41.77 seconds to find a valid plan with 1 worker, 7.55 seconds with 8 workers, 6.84 seconds with 16 workers, and just 4.92 seconds with 32 workers. Figure 5.10 shows that when the number of workers ranges from 8 to 32 the search time for parallel-CPCES steadily decreases at a consistent rate. However, with only 1 worker the search time is significantly worse compared to all other settings. We attribute this observation to the following: First, with only 1 worker the CPM cannot always select the hitting set with the highest priority for candidate plan searches, as it is forced to choose the seed hitting set each time (if all initial states have solutions). Second, there is no opportunity with one worker to explore multiple classical abstractions at each depth. Finally, with only 1 CPM worker each depth produces only one candidate plan for CTS calculation, in contrast to scenarios with multiple workers, where a valid plan can be found

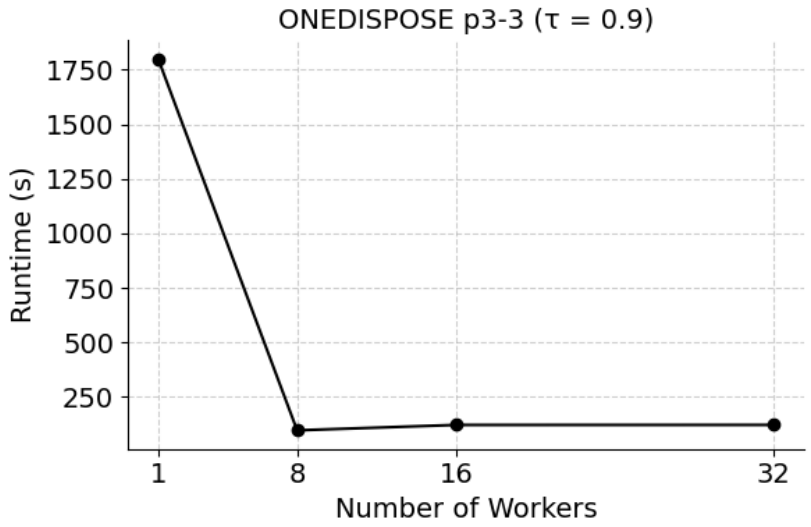


Figure 5.8: Relationship between the number of workers and runtime (seconds) for `ONEDISPOSE` p3-3 with a probability threshold of $\tau = 0.9$. The depths at which valid plans were found are: timeout (1 worker), 5 (8 workers), 8 (16 workers), and 8 (32 workers).

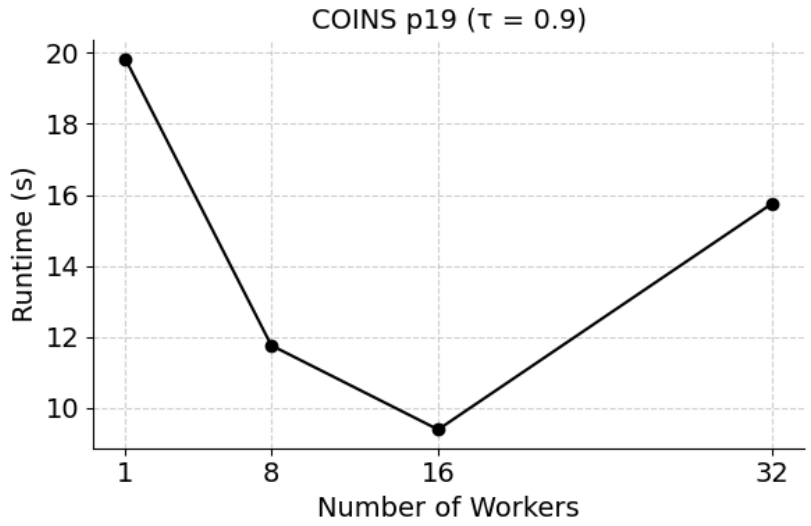


Figure 5.9: Relationship between the number of workers and runtime (seconds) for `COINS` p19 with a probability threshold of $\tau = 0.9$. The depths at which valid plans were found are: 14 (1 worker), 12 (8 workers), 9 (16 workers), and 6 (32 workers).

Figure 5.10: Relationship between the number of workers and runtime (seconds) for `DISPOSE` p4-3 with a probability threshold of $\tau = 0.9$. The depths at which valid plans were found are: 28 (1 worker), 9 (8 workers), 8 (16 workers), and 9 (32 workers).

from previous depths. In summary, with only one worker the breadth and diversity of exploration is relatively limited.

Our experimental results indicate that for most cases the best overall performance for `parallel-CPCES` occurs when the CPM is assigned 16 workers. With 32 or 8 workers performance decreases, and 1 worker yields the worst results. For example, in the `COINS` p19 problem with $\tau = 0.9$ (Figure 5.9), 16 workers find a valid plan in 9.39 seconds, 8 workers take 11.76 seconds, and 1 worker takes 19.83 seconds. Surprisingly, with 32 workers performance is worse than with 8 workers, taking 15.75 seconds to find a valid plan. Through careful analysis, we found that assigning 32 workers to the CPM triggers database-level contention in our experimental setup. Specifically, the Python + SQLite combination cannot handle high-concurrency read/write workloads efficiently, leading to locking and I/O delays that slow down inter-module communication and degrade overall performance. This observation does not contradict the overall trend — adding CPM workers improves `parallel-CPCES` 's efficiency — as evidenced by the steadily reduced search times with 1, 8, and 16 workers. While combinatorial search often

exhibits diminishing marginal returns as resources increase, in our experiments the dominant factor behind the degradation from 16 to 32 workers was the SQLite locking bottleneck rather than an intrinsic limit of the approach. Accordingly, we do not believe the scalability limits of `parallel-CPCES` have been reached.

For some rare cases, both 16 workers and 32 workers exhibit database contention issues in our experimental environment. For instance, in problem `ONEDISPOSE` p3-2 with $\tau = 0.9$ (Figure 5.8), both 16 workers and 32 workers require depth 8 to find a valid plan. A plateau appears in the graph when there are 8 to 32 workers, with search times approximately 120 seconds. This is precisely due to some database contention issues encountered with both 16 and 32 workers.

5.6.3 `parallel-CPCES` vs. `p-CPCES-hit` and `P-FF`

In this subsection, we compare the performance of `parallel-CPCES` with `p-CPCES-hit` (using the `FF` as classical planner), and with `P-FF`.

From Figure 4.7 we can see that when a problem can be solved within 1 second, `p-CPCES-hit` and `P-FF` outperform `parallel-CPCES`. However, as problems become more complex and require more than 1 second to solve, the advantages of `parallel-CPCES` become evident. This phenomenon is due to the fixed initialization time required by `parallel-CPCES` for setting up monitors, database operations, and file system reads and writes. These overheads constitute a significant portion of the total runtime for simpler problems. This explanation is also supported by the slope of the step plot. Around the 1-second mark, the slope of `parallel-CPCES` 's curve is relatively small, but it steepens sharply after crossing the 1-second threshold. This indicates that `parallel-CPCES` requires a fraction of a second to handle monitor setup, database I/O, and file system operations.

Figure 4.7 further reveals that even with only one worker assigned to the CPM, our parallel system significantly outperforms both `p-CPCES-hit` and `P-FF`. `parallel-CPCES` offers two advantages over `p-CPCES-hit`. First, `parallel-CPCES` allocates 4 workers to CTM, significantly reducing the time required for CTS computation. Second, in `parallel-CPCES`, each depth involves multiple hitting sets. There is a time gap between the generation of the first hitting set and when CPM

retrieves it from the database. During this interval the HSM can generate additional hitting sets, giving the CPM an opportunity to select the highest-priority hitting set. This prioritization increases the likelihood that `parallel-CPCES` will find a valid plan with a shallow search. This phenomenon demonstrates the substantial efficiency improvements brought by features such as the use of FCTM, ICTM, and HSM. Increasing the number of CPM workers further enhances `parallel-CPCES` 's performance. When CPM has 8 or 16 workers, the performance is nearly identical, due to occasional database contention issues when using 16 workers. However, with 32 workers the efficiency of `parallel-CPCES` noticeably declines, as database contention issues in our experimental environment occur more frequently with this higher level of concurrency. Despite these database locking challenges, `parallel-CPCES` still outperforms the single-threaded `p-CPCES-hit` and `P-FF` by a large margin. This further validates the effectiveness of our system design.

Figure 4.7 shows that `parallel-CPCES` is robust across different values of τ. No matter τ is large or small, `parallel-CPCES` performs consistently well. This adaptability can be attributed to several mechanisms: generating multiple hitting sets per depth, prioritizing hitting sets with better potential, and parallel computation of multiple candidate plans. These features mitigate the efficiency limitations that small τ values typically impose on `p-CPCES-hit`.

Our overall observations from Table 7 to Table 10:

- Problem Complexity: The more complex the problem, the greater the speed-up achieved by `parallel-CPCES` over `FF-Hit-Random`.
- Probability Threshold (τ): As τ decreases, the speed-up of `parallel-CPCES` increases significantly due to `FF-Hit-Random`'s reduced efficiency at lower thresholds.
- Number of Workers: `parallel-CPCES` with 8–32 workers consistently outperforms its single-worker counterpart, highlighting the benefits of parallelization.

From Figure 5.11a, we observe that compared with `FF-Hit-Random`, `parallel-CPCES` consistently requires shallow depths (a small number of iterations for `FF-Hit-Random`) to find a valid plan across almost all problems. Additionally,

Figure 5.11b shows that a significant number of problems can find a valid plan with a solution depth of just 1. This strongly validates the effectiveness of the mechanisms designed for HSM and CPM.

Figure 5.12 illustrate the difference between the termination depth and solution depth across various problems. The results show a clear trend: when the number of CPM workers is 8 or more, 80% of the problems have a difference of 4 or less. However, when only one CPM worker is used, the majority of problems exhibit a difference greater than 4. This discrepancy arises due to the behavior of the candidate plan management process. With multiple workers, multiple plans are generated at each depth, allowing the CTM to prioritize validating plans from shallow depths for validity. In contrast, when only one worker is used, each depth produces only one plan. Since there are two CTMs (ICTM and FCTM), the ICTM often selects plans from earlier depths for validation, leading to a larger difference between termination depth and solution depth. A potential question arises: if valid plans can be identified several depths earlier with just one worker, why does the CTM prioritize planning at greater depths? This prioritization is justified for two reasons:

1. Derivative hitting sets are generated by adding a random number of random tags to seed hitting sets. Theoretically, this means the likelihood of finding a valid plan is uniform across depths regardless of seed hitting sets. There is no inherent advantage in focusing on earlier depths for validation.
2. At low depths seed hitting sets typically have smaller sizes. This makes it less likely for derivative hitting sets in these depths to achieve the required size (large enough to cover adequate probability of satisfied initial states) for a valid plan. Consequently, plans generated in earlier depths are less likely to be valid compared to those from later depths.

5.7 Conclusions

We introduce a parallel approach to solving PCP based on `p-CPCES`, that yields superior walltime performance by leveraging multi-core CPUs. In `parallel-CPCES`, the overall process is divided into three independent components: (i) comput-

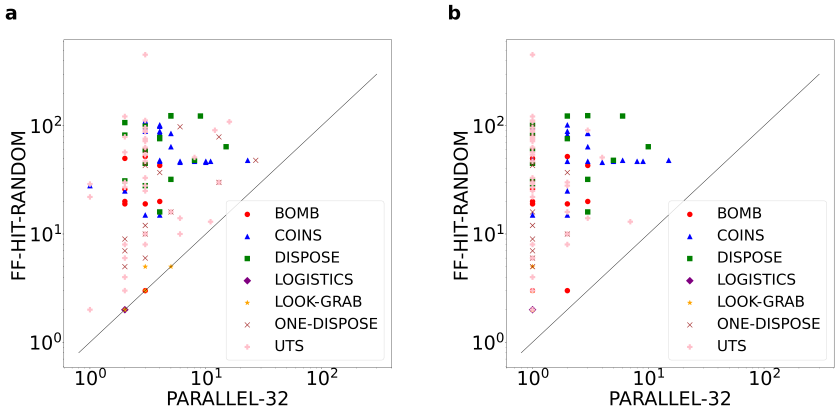


Figure 5.11: Termination depth and solution depth (iteration for `FF-Hit-Random`) used to find a valid plan by `parallel-CPCES` and `FF-Hit-Random`. Figure a shows the termination depth. Figure b shows the solution depth.

ing counter-tags, managed by the FCTM and ICTM; (ii) computing hitting sets, managed by the HSM; and (iii) searching for plans, managed by the CPM. This design allows multiple hitting sets and corresponding plans to be generated at each depth. These three components operate independently and in parallel, without interference. The execution of these components is coordinated by the MM. Inter-process communication is handled via the file system, while all intermediate and final data are stored in and retrieved from a database. To further accelerate the `parallel-CPCES` pipeline, we introduce derivative hitting sets and assign a priority to each hitting set. Plans are then computed in descending order of priority, enabling the algorithm to focus on more promising candidate plans earlier. This parallelism increases the flexibility of the algorithm and reduces the required search depth, thereby improving overall efficiency.

Experimental results demonstrate that `parallel-CPCES` can efficiently leverage multi-core CPUs to solve PCP problems, with solving speed improving as the number of workers increases. The trend is clear from 1 to 16 workers, where performance steadily improves. However, when scaling to 32 workers, efficiency decreases. This degradation is not mainly caused by algorithmic limitations or inherent diminishing returns of parallel search, but rather by the database locking bottleneck in our implementation environment. Specifically, the combination of Python and SQLite cannot handle high-concurrency read/write operations effi-

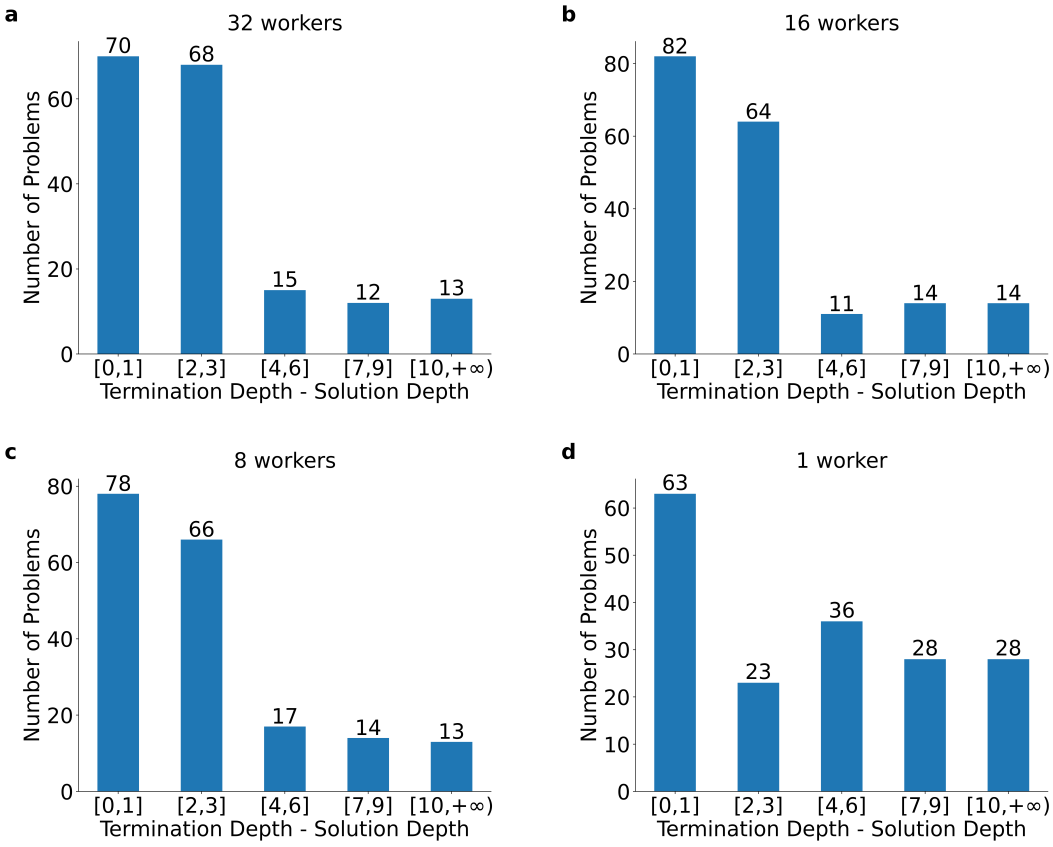


Figure 5.12: Termination vs. solution depth. Figure a shows termination vs. solution depth in CPM with 32 workers. Figure b shows termination vs. solution depth in CPM with 16 workers. Figure c shows termination vs. solution depth in CPM with 8 workers. Figure d shows termination vs. solution depth in CPM with 1 worker.

ciently, leading to contention and I/O delays that slow inter-module communication. Thus, the observed slowdown at 32 workers reflects the constraints of the experimental setup rather than a fundamental scalability limit of `parallel-CPCES`. We also compared the performance of `parallel-CPCES` with `p-CPCES` and the state-of-the-art probabilistic planner `P-FF`. The results show that `parallel-CPCES` consistently finds solutions faster than both `p-CPCES` and `P-FF`. These findings validate the effectiveness and benefit of our parallel system design.

Chapter 6

Conclusions and Future Work

In this chapter, we first summarize the overall content of this thesis. We review the three improvements to the conformant planner `CPCES` (i.e., Merging Certain-Facts, Warm-Starting CPCES, and Integrating Fast Downward into `CPCES`). We then revisit our newly proposed probabilistic conformant planners, `p-CPCES` and `p-CPCES-hit`. Finally, we discuss `parallel-CPCES`, a multi-core system for solving PCP. Building on these contributions, we also reflect on the broader potential of the counter-example based approach in planning, suggesting that it could be extended to conformant and probabilistic conformant planning with uncertain action effects, as well as to contingent planning.

6.1 Conclusions

This thesis introduces novel methods for using counter-examples to solve conformant planning and probabilistic conformant planning problems.

The first major contribution is a set of significant improvements to state-of-the-art counter-example based conformant planning approach `CPCES`. Three key enhancements were developed and evaluated:

- Merging Certain-Facts: We define certain-facts as PDDL facts with constant, unchanging values throughout plan execution. By integrating these facts into the classical planning abstraction used for candidate plan generation, we simplify the problem representation, thereby reducing computational overhead and improving efficiency.

- Warm-Starting CPCES: We introduce a *warm-start* version of CPCES, which begins with a preselected set of sample states. To maximize coverage of the problem space, we propose a strategy for selecting diverse and representative initial samples. This approach ensures that the algorithm starts with a more informed base, reducing the number of iterations required to find a valid plan compared to the baseline CPCES algorithm.
- Integration FD into CPCES: Recognizing FD's historically weaker performance compared to FF, we identify a key limitation: inefficiencies in the generation of SAS+ representations from PDDL problems. To address this, we propose an incremental method for generating SAS+ representations of problem abstracts posed by CPCES. This approach not only accelerates the conversion process but also produces more effective representations, enabling FD to perform significantly better in conformant planning tasks.

Compared with the baseline algorithm CPCES (using either FF or FD as classical planner), our experimental results confirm the effectiveness of these improvements, demonstrating a substantial improvement in overall performance.

The second major contribution is the development of a new algorithm, p-CPCES, designed to solve probabilistic conformant planning problems. In p-CPCES, a probabilistic conformant planning problem is decomposed into contexts, and a set of counter-tags is identified with their probabilities calculated efficiently using a d-DNNF representation. The algorithm iteratively constructs a composite problem, formulated as a classical planning problem, that increasingly approximates the original PCP problem with increasingly higher fidelity. In each iteration, the algorithm solves the current composite problem. If a plan is successfully generated, it is either directly applicable as a solution to the PCP problem, or is otherwise used to refine the composite problem for subsequent iterations. If no solution exists, that confirms that the original PCP problem has no solution. The composite problems posed can have large disjunctive goal terms. Classical planning systems like Madagascar and FF face significant challenges in solving composite problems involving disjunctive goals, which means that plan search is slow in practice. To overcome this, we developed p-CPCES-hit, an enhanced approach that uses a hitting set strategy. This strategy simplifies classical abstractions

by identifying and focusing on achievable subgoals for candidate plans, thereby improving the efficiency and effectiveness of the classical planning process. Our experiments demonstrate that both `p-CPCES` and `p-CPCES-hit` are effective in solving PCP problems, with `p-CPCES-hit` significantly outperforming `p-CPCES` in terms of runtime efficiency. We also observed that the choice of classical planners used in `p-CPCES` and `p-CPCES-hit` has a noticeable impact on their efficiency. Additionally, we compared the performance of `P-FF` with our counter-example based algorithms. While `P-FF` excels in solving simpler problems, `p-CPCES` and `p-CPCES-hit` are more advantageous for tackling harder problems, that take more than a few seconds to solve. We also evaluated two different hitting set strategies in `p-CPCES-hit`: minimal hitting sets and random hitting sets. Our findings reveal that random hitting sets generally lead to higher efficiency in `p-CPCES-hit`, especially for complex problems and those with lower probability thresholds.

The third major contribution is `parallel-CPCES`, a parallel system of `p-CPCES-hit` for solving PCP problems using multi-core computing systems. In this system, we use a combination of databases and a file system to facilitate communication between worker processes. In `parallel-CPCES` the available processes play specialized roles: FCTM and ICTM monitor and manage the computation of counter-tags and their probabilities, HSM handles the computation of hitting sets, CPM manages the generation of candidate plans, and MM controls the overall progress of the probabilistic conformant planning system. Each depth in `parallel-CPCES` allows for the parallel computation and generation of multiple hitting sets, multiple candidate plans, and multiple counter-tag sets. This parallelism increases the algorithm's flexibility and reduces the search depth required to find a plan, thereby improving efficiency. Experimental results demonstrate that `parallel-CPCES` effectively utilizes multiple CPUs to enhance the efficiency of `p-CPCES-hit`. For most problems, increasing the number of CPUs results in faster discovery of valid plans, validating the effectiveness of our system. However, we observed that using many CPUs can lead to IO contention and thereby slowing performance. We also compared the performance of different classical planners used in `parallel-CPCES`. The results align with those observed in `p-CPCES`, where the effectiveness of a planner depends on the specific problem. Finally, we com-

pared the efficiency of `parallel-CPCES` with `P-FF`. The results clearly show that `parallel-CPCES` significantly outperforms `P-FF` in terms of efficiency in challenging problems. The startup cost of the parallel tool means it is uncompetitive on small problems. We thereby show the promise of exploiting multi-core parallel computing environments to accelerate performance of planning under uncertainty.

6.2 Future Work

This thesis has shown that the counter-example based approach can be effectively applied to CP and PCP with uncertain initial states. A natural extension is to address problems with uncertain action effects, thereby covering the full spectrum of conformant and probabilistic conformant planning problems.

Beyond this, the counter-example based methodology can be extended to richer planning settings. One potential direction is contingent planning, where agents are allowed to make limited observations through sensing actions and can therefore pursue distinct sub-plans whose execution depends on sensing outcomes (Pryor and Collins 1996). In this setting, counter-examples include not only initial states but also observation histories, and future research could investigate how to refine strategies based on them. Another direction is multi-agent conformant planning (Li 2021), where counter-examples may correspond to initial states that prevent the achievement of a joint goal. Developing techniques to share or decompose counter-examples across multiple agents poses novel challenges.

Finally, our work on `parallel-CPCES` has demonstrated the potential of parallelizing the counter-example based methodology. A further step is to combine parallel counter-example guided search with machine learning, for example, by learning which types of counter-examples are most useful for guiding the search. Such integration could significantly improve efficiency, particularly on large-scale problems.

Here are some potential applications of counter-example based planning in real-world domains. In autonomous driving and robotics, agents often operate in partially observable environments with sensor noise or occlusions, where counter-

example based planning enables rapid reasoning over critical states, allowing for safe and efficient decision-making without the need to exhaustively consider all possible scenarios. Similarly, in disaster response and emergency rescue scenarios, where information is typically incomplete and resources are limited, counter-example based planning enables agents to efficiently generate robust action sequences by focusing only on critical states that could lead to plan failure. In healthcare and medical diagnosis, counter-examples can guide probabilistic reasoning, allowing AI systems to focus on key diagnostic possibilities rather than exhaustively enumerating all potential conditions, thereby improving the efficiency and effectiveness of treatment planning. These examples highlight the practical relevance of counter-example based approaches and motivate their continued development.

Appendix

Table 1: Performance of FF-p-CPCES in terms of runtime (seconds), number of iterations, and plan length.

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
BOMB	20-1	5.71 /20/ 37	-	-	-
BOMB	20-5	5.91 /21/ 35	-	-	-
BOMB	20-10	6.83 /21/ 30	-	-	-
BOMB	20-20	-	-	-	-
BOMB	100-1	-	-	-	-
BOMB	100-5	-	-	-	-
BOMB	100-10	-	-	-	-
BOMB	100-60	-	-	-	-
BOMB	100-100	-	-	-	-
COINS	10	2.01 /17/ 25	1.65 /17/25	-	-
COINS	12	17.78/49/ 65	15.69/49/65	-	-
COINS	15	17.42/49/ 65	-	-	-
COINS	16	38.04/49/ 83	37.26/49/83	-	-
COINS	17	43.89/49/ 91	43.8 /49/91	-	-
COINS	18	42.25/49/ 87	39.79/49/87	-	-
COINS	19	42.19/49/ 83	38.31/49/83	-	-
COINS	20	39.57/49/ 78	34.9 /49/78	-	-
COINS	21	-	-	-	-
DISPOSE	4-1	2.33 /17/ 36	-	-	-

Continued on next page

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
DISPOSE	4-2	8.67 /33/ 66	-	-	-
DISPOSE	4-3	23.86/49/100	-	-	-
DISPOSE	8-1	750.4/65/138	-	-	-
DISPOSE	8-2	-	-	-	-
DISPOSE	8-3	-	-	-	-
DISPOSE	12-1	-	-	-	-
DISPOSE	12-2	-	-	-	-
DISPOSE	12-3	-	-	-	-
LOGISTICS	1	-	-	-	-
LOGISTICS	2	-	-	-	-
LOGISTICS	3	-	-	-	-
LOGISTICS	4	-	-	-	-
LOOK-GRAB	4-1-1	1.36 / 6 / 12	2.09 / 7 /12	2.83 / 6 / 8	1.03/2/4
LOOK-GRAB	4-1-2	0.51 / 2 / 4	0.51 / 2 / 4	0.65 / 2 / 4	0.84/2/4
LOOK-GRAB	4-1-3	0.51 / 2 / 4	0.54 / 2 / 4	0.67 / 2 / 4	0.86/2/4
LOOK-GRAB	4-2-1	-	-	-	-
LOOK-GRAB	4-2-2	21.24/ 2 / 4	-	-	-
LOOK-GRAB	4-2-3	21.13/ 2 / 4	-	-	-
LOOK-GRAB	4-3-1	-	-	-	-
LOOK-GRAB	4-3-2	-	-	-	-
LOOK-GRAB	4-3-3	-	-	-	-
LOOK-GRAB	8-1-1	-	-	-	-
LOOK-GRAB	8-1-2	-	-	-	-
LOOK-GRAB	8-1-3	-	-	-	-
LOOK-GRAB	8-2-1	-	-	-	-
LOOK-GRAB	8-2-2	-	-	-	-
LOOK-GRAB	8-2-3	-	-	-	-
LOOK-GRAB	8-3-1	-	-	-	-

Continued on next page

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
LOOK-GRAB	8-3-2	-	-	-	-
LOOK-GRAB	8-3-3	-	-	-	-
ONEDISPOSE	2-2	0.84 / 7 / 23	4.27 /14/22	-	-
ONEDISPOSE	2-3	3.25 / 9 / 48	-	-	-
ONEDISPOSE	3-2	8.59 /16/ 88	-	-	-
ONEDISPOSE	3-3	-	-	-	-
ONEDISPOSE	4-2	-	-	-	-
ONEDISPOSE	4-3	-	-	-	-
ONEDISPOSE	5-2	-	-	-	-
ONEDISPOSE	5-3	-	-	-	-
ONEDISPOSE	6-2	-	-	-	-
ONEDISPOSE	6-3	-	-	-	-
UTS	1	0.14 / 3 / 4	0.13 / 3 / 4	0.13 / 3 / 4	0.14/2/2
UTS	2	0.27 / 5 / 10	0.27 / 5 /12	0.58 / 7 / 8	0.54/5/8
UTS	3	0.51 / 7 / 16	0.47 / 7 /16	69.76/16/18	-
UTS	4	0.7 / 9 / 22	0.74 / 9 /28	-	-
UTS	5	1.03 /11/ 28	1.11 /11/28	-	-
UTS	6	1.65 /13/ 34	-	-	-
UTS	7	2.28 /15/ 40	-	-	-
UTS	8	3.07 /17/ 46	-	-	-
UTS	9	4.09 /19/ 52	-	-	-
UTS	20	6.1 /21/ 58	-	-	-
UTS	30	19.58/31/ 88	-	-	-
UTS	40	58.17/41/118	-	-	-
UTS	50	-	-	-	-
UTS	60	-	-	-	-

Table 2: Performance of `FF-Hit-Random` in terms of runtime (seconds), number of iterations, and plan length.

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
BOMB	20-1	7.4 /19/ 37	24.83/ 50 / 39	13.41/ 19 / 29	2.06 / 2 / 19
BOMB	20-5	7.52 /20/ 35	28.28/ 52 / 33	15.95/ 20 / 29	1.99 / 2 / 15
BOMB	20-10	7.47 /19/ 30	18.95/ 43 / 28	16.85/ 26 / 20	1.94 / 2 / 14
BOMB	20-20	-	-	-	1.84 / 2 / 12
BOMB	100-1	-	-	-	207.6/ 3 /107
BOMB	100-5	-	-	-	186.4/ 3 /105
BOMB	100-10	-	-	-	189.5/ 3 /100
BOMB	100-60	-	-	-	103 / 3 / 52
BOMB	100-100	-	-	-	112.8/ 3 / 50
COINS	10	1.69 /15/ 25	1.58 / 15 / 25	4.73 / 28 / 25	4.61 / 25 / 23
COINS	12	19.52/47/ 65	18.97/ 48 / 65	63.28/102/ 63	61.83/ 80 / 62
COINS	15	19.06/46/ 65	96.57/ 86 / 60	347.5/ 64 / 54	63.8 / 15 / 17
COINS	16	41.42/47/ 83	39.99/ 47 / 83	106.5/ 99 / 82	103.1/ 85 / 77
COINS	17	46.35/47/ 91	46.6 / 48 / 91	121 / 97 / 88	119.4/ 89 / 87
COINS	18	44.16/47/ 87	44.7 / 46 / 87	135.7/111/ 85	124.1/ 97 / 83
COINS	19	42.84/48/ 83	41.49/ 47 / 83	114.1/102/ 81	110.5/ 89 / 80
COINS	20	38.88/47/ 78	38.8 / 47 / 78	102.9/101/ 76	102.7/ 83 / 73
COINS	21	-	-	-	-
DISPOSE	4-1	2.53 /16/ 36	6.12 / 28 / 38	9.79 / 31 / 31	57.17/107/ 30
DISPOSE	4-2	10.16/32/ 66	68.18/124/ 66	48.41/ 82 / 58	40.99/ 45 / 56
DISPOSE	4-3	28.06/48/100	129.1/123/ 97	94.7 / 99 / 94	112.9/ 76 / 73
DISPOSE	8-1	766.3/64/138	1432 /123/137	873.6/ 78 /121	741.3/ 60 /109
DISPOSE	8-2	-	-	-	-
DISPOSE	8-3	-	-	-	-
DISPOSE	12-1	-	-	-	-
DISPOSE	12-2	-	-	-	-

Continued on next page

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
DISPOSE	12-3	-	-	-	-
LOGISTICS	1	-	-	-	-
LOGISTICS	2	-	-	-	-
LOGISTICS	3	-	-	-	161 / 2 / 9
LOGISTICS	4	-	-	-	-
LOOK-GRAB	4-1-1	1.01 / 5 / 12	1.14 / 5 / 12	1.1 / 3 / 10	1.02 / 2 / 8
LOOK-GRAB	4-1-2	0.48 / 2 / 4	0.51 / 2 / 4	0.63 / 2 / 4	0.82 / 2 / 4
LOOK-GRAB	4-1-3	0.5 / 2 / 4	0.53 / 2 / 4	0.65 / 2 / 4	0.84 / 2 / 4
LOOK-GRAB	4-2-1	-	-	-	-
LOOK-GRAB	4-2-2	20.06/ 2 / 4	22.36/ 2 / 4	29 / 2 / 4	46.74/ 2 / 4
LOOK-GRAB	4-2-3	20.23/ 2 / 4	22.44/ 2 / 4	28.94/ 2 / 4	46.59/ 2 / 4
LOOK-GRAB	4-3-1	-	-	-	-
LOOK-GRAB	4-3-2	-	-	-	-
LOOK-GRAB	4-3-3	-	-	-	-
LOOK-GRAB	8-1-1	-	-	-	-
LOOK-GRAB	8-1-2	-	-	-	-
LOOK-GRAB	8-1-3	-	-	-	-
LOOK-GRAB	8-2-1	-	-	-	-
LOOK-GRAB	8-2-2	-	-	-	-
LOOK-GRAB	8-2-3	-	-	-	-
LOOK-GRAB	8-3-1	-	-	-	-
LOOK-GRAB	8-3-2	-	-	-	-
LOOK-GRAB	8-3-3	-	-	-	-
ONEDISPOSE	2-2	0.93 / 9 / 32	0.9 / 6 / 23	2 / 7 / 22	2.01 / 5 / 19
ONEDISPOSE	2-3	3.29 /10/ 48	7.94 / 12 / 46	13.57/ 12 / 46	24.84/ 12 / 35
ONEDISPOSE	3-2	8.8 /16/ 89	51.38/ 37 / 80	90.95/ 43 / 95	212.8/ 59 / 95
ONEDISPOSE	3-3	401.6/30/160	-	-	-
ONEDISPOSE	4-2	330.4/48/210	696.1/ 79 /217	1508 / 98 /211	-

Continued on next page

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
ONEDISPOSE	4-3	-	-	-	-
ONEDISPOSE	5-2	-	-	-	-
ONEDISPOSE	5-3	-	-	-	-
ONEDISPOSE	6-2	-	-	-	-
ONEDISPOSE	6-3	-	-	-	-
UTS	1	0.09 / 2 / 4	0.1 / 2 / 4	0.09 / 2 / 4	0.12 / 2 / 4
UTS	2	0.25 / 4 / 10	0.24 / 4 / 10	0.26 / 3 / 11	0.23 / 2 / 7
UTS	3	0.42 / 6 / 16	0.47 / 6 / 16	0.82 / 8 / 14	0.51 / 3 / 10
UTS	4	0.71 / 8 / 28	0.68 / 8 / 22	3.62 / 22 / 18	1.85 / 8 / 23
UTS	5	1.05 /10/ 28	1 / 10 / 28	12.75/ 57 / 26	9.68 / 30 / 21
UTS	6	1.78 /13/ 34	4.51 / 25 / 34	24.08/ 79 / 34	54.88/122/ 27
UTS	7	2.29 /14/ 40	5.88 / 28 / 38	8.01 / 29 / 38	298.8/454/ 26
UTS	8	3.17 /16/ 46	9.11 / 30 / 60	12.62/ 33 / 46	79.8 /112/ 46
UTS	9	4.5 /18/ 52	20.02/ 51 / 50	12.01/ 27 / 44	75.01/ 91 / 42
UTS	20	6.01 /20/ 58	31.62/ 65 / 56	27.64/ 47 / 52	88 / 88 / 58
UTS	30	19.76/30/ 88	89.29/ 91 / 86	67.5 / 54 / 99	170.6/ 76 / 93
UTS	40	56.83/41/118	195.9/109/114	201.8/ 73 /130	413.4/ 96 / 88
UTS	50	-	-	361.3/ 72 /128	579.8/ 63 /128
UTS	60	-	-	-	-

Table 3: Performance of `FF-Hit-Minimal` in terms of runtime (seconds), number of iterations, and plan length.

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
BOMB	20-1	3.22 /10/ 37	3.22 / 5 /37	8.34 / 10 / 29	1.97 / 2 / 23
BOMB	20-5	4.06 /11/ 35	3.23 / 5 /33	2.53 / 3 / 25	1.99 / 2 / 21
BOMB	20-10	4.11 /12/ 30	2.83 / 5 /28	6.57 / 8 / 20	1.88 / 2 / 12
BOMB	20-20	-	396.6/213/19	1667 / 25 / 15	1.81 / 2 / 9
BOMB	100-1	-	-	-	-
BOMB	100-10	-	-	-	-
BOMB	100-100	437.6/28/ 99	-	-	-
BOMB	100-5	-	-	-	-
BOMB	100-60	-	-	-	-
COINS	10	1.59 /17/ 25	1.62 / 17 /25	22 /121/ 24	147.8/561/ 23
COINS	12	16.2 /49/ 65	15.27/ 49 /65	-	-
COINS	15	15.56/49/ 65	-	-	43.99/ 14 / 16
COINS	16	35.74/49/ 83	37.56/ 49 /83	-	-
COINS	17	42.91/49/ 91	40.75/ 49 /91	-	-
COINS	18	40.64/49/ 87	41.13/ 49 /87	-	-
COINS	19	37.43/49/ 83	38.91/ 49 /83	-	-
COINS	20	36.39/49/ 78	34.33/ 49 /78	-	-
COINS	21	-	-	-	-
DISPOSE	4-1	2.23 /17/ 36	30.27/121/35	-	-
DISPOSE	4-2	8.68 /33/ 70	361.3/497/68	-	-
DISPOSE	4-3	23.04/49/104	-	-	-
DISPOSE	8-1	698.6/65/138	-	-	-
DISPOSE	8-2	-	-	-	-
DISPOSE	8-3	-	-	-	-
DISPOSE	12-1	-	-	-	-
DISPOSE	12-2	-	-	-	-

Continued on next page

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
DISPOSE	12-3	-	-	-	-
LOGISTICS	1	-	-	-	-
LOGISTICS	2	-	-	-	-
LOGISTICS	3	-	-	-	177.5/ 4 / 9
LOGISTICS	4	-	-	-	-
LOOK-GRAB	4-1-1	0.95 / 6 / 12	1.08 / 4 /10	1.09 / 3 / 8	0.99 / 2 / 6
LOOK-GRAB	4-1-2	0.47 / 2 / 4	0.51 / 2 / 4	0.61 / 2 / 4	0.81 / 2 / 4
LOOK-GRAB	4-1-3	0.47 / 2 / 4	0.51 / 2 / 4	0.62 / 2 / 4	0.8 / 2 / 4
LOOK-GRAB	4-2-1	-	-	-	-
LOOK-GRAB	4-2-2	19.19/ 2 / 4	22 / 2 / 4	29.76/ 2 / 4	45.41/ 2 / 4
LOOK-GRAB	4-2-3	19.23/ 2 / 4	21.74/ 2 / 4	29.17/ 2 / 4	45.23/ 2 / 4
LOOK-GRAB	4-3-1	-	-	-	-
LOOK-GRAB	4-3-2	-	-	-	-
LOOK-GRAB	4-3-3	-	-	-	-
LOOK-GRAB	8-1-1	-	-	-	-
LOOK-GRAB	8-1-2	-	-	-	-
LOOK-GRAB	8-1-3	-	-	-	-
LOOK-GRAB	8-2-1	-	-	-	-
LOOK-GRAB	8-2-2	-	-	-	-
LOOK-GRAB	8-2-3	-	-	-	-
LOOK-GRAB	8-3-1	-	-	-	-
LOOK-GRAB	8-3-2	-	-	-	-
LOOK-GRAB	8-3-3	-	-	-	-
ONEDISPOSE	2-2	0.72 / 7 / 29	1.43 / 10 /28	3.48 / 15 / 19	6.6 / 19 / 18
ONEDISPOSE	2-3	2.89 / 7 / 48	13.49/ 25 /48	36.99/ 37 / 41	133 / 71 / 32
ONEDISPOSE	3-2	8.74 /17/ 84	1251 / 89 /69	-	-
ONEDISPOSE	3-3	386.5/38/160	-	-	-
ONEDISPOSE	4-2	143.5/38/188	-	-	-

Continued on next page

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
ONEDISPOSE	4-3	-	-	-	-
ONEDISPOSE	5-2	-	-	-	-
ONEDISPOSE	5-3	-	-	-	-
ONEDISPOSE	6-2	-	-	-	-
ONEDISPOSE	6-3	-	-	-	-
UTS	1	0.13 / 3 / 4	0.12 / 3 / 4	0.12 / 3 / 4	0.12 / 2 / 2
UTS	2	0.26 / 5 / 12	0.24 / 5 /10	0.57 / 7 / 8	0.54 / 5 / 6
UTS	3	0.42 / 7 / 16	0.41 / 7 /16	1.6 / 16 / 14	2.48 / 16 / 11
UTS	4	0.7 / 9 / 22	0.67 / 9 /22	9.51 / 57 / 23	12.71/ 57 / 14
UTS	5	1.04 /11/ 28	0.95 / 11 /28	26.3 /121/ 24	66.65/211/ 18
UTS	6	1.4 /13/ 34	13.53/ 67 /42	22.95/ 12 /223	351.9/793/ 32
UTS	7	2.04 /15/ 40	23.68/ 92 /38	66.74/ 16 /315	-
UTS	8	2.86 /17/ 46	40.49/121/44	135.2/ 15 /559	-
UTS	9	3.72 /19/ 52	64.85/154/50	249.3/ 18 /644	202 / 9 /648
UTS	20	5.57 /21/ 58	105.8/191/56	-	-
UTS	30	16.81/31/ 88	-	-	-
UTS	40	49.28/41/118	-	-	-
UTS	50	-	-	-	-
UTS	60	-	-	-	-

Table 4: Performance of `Mad-p-CPCES` in terms of runtime (seconds), number of iterations, and plan length.

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
BOMB	20-1	11.91/20/ 37	-	-	-
BOMB	20-5	6.88 /21/ 35	-	1170 /69/ 25	727.4/51/ 29
BOMB	20-10	6.46 /21/ 35	23.83/32/ 28	328.9/24/ 25	4.81 / 2 / 12
BOMB	20-20	-	-	-	3.75 / 2 / 21
BOMB	100-1	-	-	-	-
BOMB	100-5	-	-	-	-
BOMB	100-10	-	-	-	-
BOMB	100-60	-	-	-	-
BOMB	100-100	-	-	-	-
COINS	10	2.5 /16/ 39	2.7 /17/ 42	14.11/35/ 30	27.88/47/ 27
COINS	12	45.63/48/ 90	49.91/49/ 85	-	-
COINS	15	63.66/48/137	-	-	-
COINS	16	-	-	-	-
COINS	17	-	-	-	-
COINS	18	-	-	-	-
COINS	19	-	-	-	-
COINS	20	1582 /49/154	1551 /49/148	-	-
COINS	21	-	-	-	-
DISPOSE	4-1	27.33/17/ 42	-	-	-
DISPOSE	4-2	115.7/33/ 54	-	-	-
DISPOSE	4-3	270.9/49/ 74	-	-	-
DISPOSE	8-1	-	-	-	-
DISPOSE	8-2	-	-	-	-
DISPOSE	8-3	-	-	-	-
DISPOSE	12-1	-	-	-	-
DISPOSE	12-2	-	-	-	-

Continued on next page

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
DISPOSE	12-3	-	-	-	-
LOGISTICS	1	-	-	-	-
LOGISTICS	2	-	-	-	-
LOGISTICS	3	-	-	-	-
LOGISTICS	4	-	-	-	-
LOOK-GRAB	4-1-1	1.98 / 7 / 18	108.7/ 6 / 10	77.14/ 9 / 12	2.27 / 3 / 4
LOOK-GRAB	4-1-2	0.57 / 2 / 5	0.63 / 2 / 4	1.2 / 2 / 8	1.36 / 2 / 4
LOOK-GRAB	4-1-3	0.59 / 2 / 7	0.65 / 2 / 4	1 / 2 / 5	1.33 / 2 / 4
LOOK-GRAB	4-2-1	-	-	-	-
LOOK-GRAB	4-2-2	21.54/ 2 / 4	-	-	-
LOOK-GRAB	4-2-3	21.77/ 2 / 4	-	-	-
LOOK-GRAB	4-3-1	-	-	-	-
LOOK-GRAB	4-3-2	-	-	-	-
LOOK-GRAB	4-3-3	-	-	-	-
LOOK-GRAB	8-1-1	-	-	-	-
LOOK-GRAB	8-1-2	-	-	-	-
LOOK-GRAB	8-1-3	-	-	-	-
LOOK-GRAB	8-2-1	-	-	-	-
LOOK-GRAB	8-2-2	-	-	-	-
LOOK-GRAB	8-2-3	-	-	-	-
LOOK-GRAB	8-3-1	-	-	-	-
LOOK-GRAB	8-3-2	-	-	-	-
LOOK-GRAB	8-3-3	-	-	-	-
ONEDISPOSE	2-2	1.26 / 9 / 20	351.5/29/ 19	820.1/46/ 34	435.1/55/ 33
ONEDISPOSE	2-3	5.97 /18/ 33	-	-	-
ONEDISPOSE	3-2	206.8/34/ 52	-	-	-
ONEDISPOSE	3-3	-	-	-	-
ONEDISPOSE	4-2	-	-	-	-

Continued on next page

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
ONEDISPOSE	4-3	-	-	-	-
ONEDISPOSE	5-2	-	-	-	-
ONEDISPOSE	5-3	-	-	-	-
ONEDISPOSE	6-2	-	-	-	-
ONEDISPOSE	6-3	-	-	-	-
UTS	1	0.22 / 3 / 7	0.22 / 3 / 7	0.25 / 3 / 7	0.19 / 2 / 2
UTS	2	0.49 / 5 / 13	0.46 / 5 / 15	0.95 / 6 / 11	0.76 / 4 / 9
UTS	3	1.04 / 7 / 45	1.01 / 7 / 48	2.76 /10/ 42	2 / 6 / 26
UTS	4	2.37 / 9 / 83	2.52 / 9 / 94	10.27/ 8 / 43	3.69 / 5 / 33
UTS	5	6.33 /11/125	6.25 /11/167	64.87/ 6 / 98	31.14/ 4 /104
UTS	6	24.79/13/292	473.7/ 7 /155	213.4/ 5 /179	137.9/ 3 /124
UTS	7	81.69/13/501	-	862.2/ 5 /187	306.1/ 3 /167
UTS	8	483.7/16/708	-	-	-
UTS	9	-	-	-	-
UTS	20	-	-	-	-
UTS	30	-	-	-	-
UTS	40	-	-	-	-
UTS	50	-	-	-	-
UTS	60	-	-	-	-

Table 5: Performance of `Mad-Hit-Random` in terms of runtime (seconds), number of iterations, and plan length.

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
BOMB	20-1	20.15/19/ 37	66.13/ 50 / 37	23.52/ 21 / 29	3.61 / 3 / 19
BOMB	20-5	10.26/19/ 35	34.82/ 56 / 38	15.61/ 18 / 25	2.28 / 2 / 18
BOMB	20-10	10.2 /19/ 32	29.31/ 42 / 32	21.56/ 25 / 26	2.07 / 2 / 10
BOMB	20-20	-	-	-	2.24 / 2 / 20
BOMB	100-1	-	-	-	-
BOMB	100-5	-	-	-	223.6/ 3 / 99
BOMB	100-10	-	-	-	212.1/ 3 /102
BOMB	100-60	-	-	-	-
BOMB	100-100	-	-	-	-
COINS	10	2.21 /14/ 42	2.72 / 14 / 42	4.99 / 22 / 40	6 /24/ 40
COINS	12	50.36/47/117	53.27/ 47 /112	128.3/102/169	113.9/84/125
COINS	15	67.52/45/110	138 / 74 / 93	396.3/ 74 /101	59.33/15/ 39
COINS	16	-	-	-	-
COINS	17	-	-	-	-
COINS	18	-	-	-	-
COINS	19	-	-	-	-
COINS	20	-	1783 / 47 /147	-	1494 /92/156
COINS	21	-	-	-	-
DISPOSE	4-1	34.42/16/ 39	59.38/ 30 / 35	35.81/ 32 / 35	51.15/50/ 34
DISPOSE	4-2	138.8/32/ 52	747.4/127/ 61	176.4/ 52 / 50	87.22/45/ 58
DISPOSE	4-3	290 /47/ 75	900.5/126/ 75	437.1/ 95 / 80	305.9/75/ 69
DISPOSE	8-1	-	-	-	-
DISPOSE	8-2	-	-	-	-
DISPOSE	8-3	-	-	-	-
DISPOSE	12-1	-	-	-	-
DISPOSE	12-2	-	-	-	-

Continued on next page

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
DISPOSE	12-3	-	-	-	-
LOGISTICS	1	185.7/ 9 / 48	179 / 8 / 49	175.5/ 9 / 48	473.9/22/ 48
LOGISTICS	2	29.65/ 5 / 35	28.73/ 6 / 35	26.5 / 5 / 35	24.37/ 7 / 32
LOGISTICS	3	1.14 / 3 / 23	1.12 / 3 / 23	0.95 / 3 / 23	0.5 / 2 / 27
LOGISTICS	4	-	-	-	-
LOOK-GRAB	4-1-1	2.39 / 7 / 12	1.13 / 4 / 12	1.09 / 3 / 11	1.03 / 2 / 10
LOOK-GRAB	4-1-2	0.56 / 2 / 5	0.6 / 2 / 5	0.69 / 2 / 5	0.9 / 2 / 6
LOOK-GRAB	4-1-3	0.6 / 2 / 6	0.62 / 2 / 6	0.71 / 2 / 7	0.93 / 2 / 6
LOOK-GRAB	4-2-1	-	-	-	-
LOOK-GRAB	4-2-2	20.18/ 2 / 7	22.6 / 2 / 7	29.47/ 2 / 5	47.71/ 2 / 5
LOOK-GRAB	4-2-3	20.29/ 2 / 7	23.26/ 2 / 7	30.05/ 2 / 7	48.17/ 2 / 7
LOOK-GRAB	4-3-1	-	-	-	-
LOOK-GRAB	4-3-2	-	-	-	-
LOOK-GRAB	4-3-3	-	-	-	-
LOOK-GRAB	8-1-1	-	-	-	154.2/ 3 / 28
LOOK-GRAB	8-1-2	-	1727 / 10 / 45	272.5/ 11 / 35	21.95/ 2 / 23
LOOK-GRAB	8-1-3	-	-	-	-
LOOK-GRAB	8-2-1	-	-	-	-
LOOK-GRAB	8-2-2	-	-	-	-
LOOK-GRAB	8-2-3	-	-	-	-
LOOK-GRAB	8-3-1	-	-	-	-
LOOK-GRAB	8-3-2	-	-	-	-
LOOK-GRAB	8-3-3	-	-	-	-
ONEDISPOSE	2-2	1.83 /10/ 20	2.06 / 11 / 22	3.74 / 13 / 18	4.45 /10/ 16
ONEDISPOSE	2-3	8.09 /21/ 34	21.09/ 24 / 52	34.28/ 27 / 35	53.57/25/ 39
ONEDISPOSE	3-2	256.6/39/ 66	167.7/ 60 / 53	105.5/ 44 / 68	327.6/84/ 51
ONEDISPOSE	3-3	-	-	-	-
ONEDISPOSE	4-2	-	-	-	-

Continued on next page

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
ONEDISPOSE	4-3	-	-	-	-
ONEDISPOSE	5-2	-	-	-	-
ONEDISPOSE	5-3	-	-	-	-
ONEDISPOSE	6-2	-	-	-	-
ONEDISPOSE	6-3	-	-	-	-
UTS	1	0.15 / 2 / 7	0.15 / 2 / 7	0.15 / 2 / 7	0.17 / 2 / 7
UTS	2	0.45 / 4 / 14	0.45 / 4 / 14	0.38 / 3 / 14	0.29 / 2 / 10
UTS	3	1 / 6 / 45	1.08 / 6 / 45	1.57 / 8 / 44	0.76 / 3 / 39
UTS	4	2.43 / 8 / 81	2.51 / 8 / 81	2.14 / 6 / 82	1.4 / 3 / 87
UTS	5	6.36 /10/144	6.32 / 10 /144	11.31/ 14 /158	4.21 / 5 /112
UTS	6	17.85/10/307	26.87/ 11 /241	18.96/ 12 /266	15.41/ 7 /224
UTS	7	115.8/13/387	89.96/ 13 /440	52.67/ 14 /286	26.52/ 7 /291
UTS	8	570.1/15/681	405.2/ 16 /736	136.1/ 17 /563	64.73/ 7 /583
UTS	9	-	-	304.8/ 19 /746	152.3/10/767
UTS	20	-	-	429.9/ 14 /812	97.97/ 6 /574
UTS	30	-	-	-	-
UTS	40	-	-	-	-
UTS	50	-	-	-	-
UTS	60	-	-	-	-

Table 6: Performance of `Mad-Hit-Minimal` in terms of runtime (seconds), number of iterations, and plan length.

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
BOMB	p20-1	10.25/10/ 37	8.51 / 6 / 37	8.83 / 7 / 29	2.45 / 2 / 22
BOMB	p20-5	6.26 /12/ 35	4.68 / 7 / 33	4.81 / 5 / 25	2.22 / 2 / 26
BOMB	p20-10	7.33 /12/ 35	5.06 / 7 / 36	4.01 / 4 / 27	2.14 / 2 / 10
BOMB	p20-20	-	-	-	2.13 / 2 / 20
BOMB	p100-1	-	-	-	-
BOMB	p100-5	-	-	-	-
BOMB	p100-10	-	-	-	-
BOMB	p100-60	-	-	-	-
BOMB	p100-100	-	-	-	-
COINS	p10	2.14 /14/ 40	2.6 /17/ 42	13.58/ 61 / 40	23.02/90/ 35
COINS	p12	45.43/49/ 97	44.1 /49/ 85	-	-
COINS	p15	60.21/48/108	-	24.45/ 15 / 46	16.81/12/ 46
COINS	p16	-	-	-	-
COINS	p17	-	-	-	-
COINS	p18	-	-	-	-
COINS	p19	-	-	-	-
COINS	p20	756.7/49/149	953.4/49/163	-	-
COINS	p21	-	-	-	-
DISPOSE	p-4-1	23.94/17/ 36	235.8/95/ 41	158.2/163/ 46	55.61/81/ 27
DISPOSE	p-4-2	101.1/33/ 71	-	-	-
DISPOSE	p-4-3	299.4/48/ 88	-	-	-
DISPOSE	p-8-1	-	-	-	-
DISPOSE	p-8-2	-	-	-	-
DISPOSE	p-8-3	-	-	-	-
DISPOSE	p-12-1	-	-	-	-
DISPOSE	p-12-2	-	-	-	-

Continued on next page

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
DISPOSE	p-12-3	-	-	-	-
LOGISTICS	p1	175.4/10/ 48	163.3/10/ 52	166.5/ 10 / 48	724.5/37/ 41
LOGISTICS	p2	29.23/ 7 / 37	25.5 / 7 / 37	25.2 / 7 / 36	43.46/16/ 33
LOGISTICS	p3	1.19 / 4 / 24	1.17 / 4 / 24	1.17 / 4 / 24	0.84 / 4 / 25
LOGISTICS	p4	-	-	-	-
LOOK-GRAB	p-4-1-1	1.2 / 5 / 12	0.85 / 3 / 10	1.48 / 4 / 10	1.03 / 2 / 7
LOOK-GRAB	p-4-1-2	0.54 / 2 / 5	0.56 / 2 / 5	0.67 / 2 / 5	0.85 / 2 / 5
LOOK-GRAB	p-4-1-3	0.56 / 2 / 7	0.57 / 2 / 7	0.7 / 2 / 7	0.88 / 2 / 7
LOOK-GRAB	p-4-2-1	-	-	-	-
LOOK-GRAB	p-4-2-2	20.62/ 2 / 7	22.27/ 2 / 7	29.3 / 2 / 7	47.3 / 2 / 7
LOOK-GRAB	p-4-2-3	20.51/ 2 / 7	22.4 / 2 / 7	29.28/ 2 / 7	47.09/ 2 / 7
LOOK-GRAB	p-4-3-1	-	-	-	-
LOOK-GRAB	p-4-3-2	-	-	-	-
LOOK-GRAB	p-4-3-3	-	-	-	-
LOOK-GRAB	p-8-1-1	-	-	-	148 /16/ 34
LOOK-GRAB	p-8-1-2	-	1289 /21/ 52	194.8/ 18 / 22	59.07/ 7 / 22
LOOK-GRAB	p-8-1-3	-	-	-	-
LOOK-GRAB	p-8-2-1	-	-	-	-
LOOK-GRAB	p-8-2-2	-	-	-	-
LOOK-GRAB	p-8-2-3	-	-	-	-
LOOK-GRAB	p-8-3-1	-	-	-	-
LOOK-GRAB	p-8-3-2	-	-	-	-
LOOK-GRAB	p-8-3-3	-	-	-	-
ONEDISPOSE	p-2-2	1.2 / 9 / 21	1.58 /10/ 22	4.44 / 17 / 21	4.34 /11/ 21
ONEDISPOSE	p-2-3	4.69 /13/ 38	21.87/35/ 33	75.4 / 60 / 36	122.5/62/ 37
ONEDISPOSE	p-3-2	103.2/26/ 57	-	-	-
ONEDISPOSE	p-3-3	-	-	-	-
ONEDISPOSE	p-4-2	-	-	-	-

Continued on next page

Domain	Instance	Runtime(s) / Iterations / Plan Length			
		$\tau = .99$	$\tau = .9$	$\tau = .75$	$\tau = .5$
ONEDISPOSE	p-4-3	-	-	-	-
ONEDISPOSE	p-5-2	-	-	-	-
ONEDISPOSE	p-5-3	-	-	-	-
ONEDISPOSE	p-6-2	-	-	-	-
ONEDISPOSE	p-6-3	-	-	-	-
UTS	p1	0.22 / 3 / 7	0.22 / 3 / 7	0.21 / 3 / 7	0.17 / 2 / 5
UTS	p2	0.47 / 5 / 14	0.48 / 5 / 13	0.85 / 7 / 14	0.73 / 5 / 12
UTS	p3	0.9 / 7 / 43	1.08 / 7 / 47	3.2 / 16 / 44	4 /16/ 40
UTS	p4	2 / 9 / 82	2.13 / 9 / 93	5.59 / 15 / 97	2.86 / 7 / 89
UTS	p5	5.27 /10/139	5.45 /10/128	7.66 / 13 /145	5.24 / 6 /133
UTS	p6	19.23/12/302	16.14/14/245	18.54/ 14 /278	14.14/ 9 /225
UTS	p7	68.07/12/394	56.75/19/341	45.79/ 16 /440	28.99/11/206
UTS	p8	466 /15/577	182.5/19/712	125.7/ 26 /461	40.43/ 5 /575
UTS	p9	-	727.6/24/581	225.4/ 26 /595	160.8/13/817
UTS	p20	-	-	443.5/ 28 /668	168.6/ 6 /892
UTS	p30	-	-	-	-
UTS	p40	-	-	-	-
UTS	p50	-	-	-	-
UTS	p60	-	-	-	-

Table 7: Speed-up factors of `parallel-CPCES` over `FF-Hit-Random` at $\tau = 0.99$. Each ratio-x column represents the speed-up achieved by `parallel-CPCES` with x workers relative to `FF-Hit-Random`.

Domain	Instance	Ratio-32	Ratio-16	Ratio-8	Ratio-1
BOMB	20-1	6.12	3.61	5.48	1.09
BOMB	20-5	3.55	5.3	3.36	0.817
BOMB	20-10	6.5	3.44	3.29	0.546
BOMB	20-20	-	-	-	-
BOMB	100-1	9/0	9/0	9/0	9/0
BOMB	100-5	9/0	9/0	9/0	9/0
BOMB	100-10	9/0	9/0	9/0	9/0
BOMB	100-60	9/0	9/0	9/0	9/0
BOMB	100-100	9/0	9/0	9/0	9/0
COINS	10	2.06	2.22	1.07	0.583
COINS	12	3.28	2.17	2.56	1.38
COINS	15	2.92	2.53	1.43	1.42
COINS	16	8.76	5.64	2.98	3
COINS	17	3.51	6.36	3.82	2.95
COINS	18	3.51	4.36	5.33	2.04
COINS	19	4.3	2.39	3.9	2.15
COINS	20	4.15	5.59	5.15	1.21
COINS	21	-	-	-	-
DISPOSE	4-1	1.45	1.42	0.894	0.405
DISPOSE	4-2	3.64	2.71	1.63	0.936
DISPOSE	4-3	4.05	2.2	2.93	1.24
DISPOSE	8-1	10.3	15.1	16.3	11.6
DISPOSE	8-2	9/0	9/0	9/0	9/0
DISPOSE	8-3	9/0	9/0	9/0	8/0
DISPOSE	12-1	9/0	9/0	9/0	9/0
DISPOSE	12-2	-	-	-	-

Continued on next page

Domain	Instance	Ratio-32	Ratio-16	Ratio-8	Ratio-1
DISPOSE	12-3	-	-	-	-
LOGISTICS	1	-	-	-	-
LOGISTICS	2	-	-	-	-
LOGISTICS	3	-	-	-	-
LOGISTICS	4	-	-	-	-
LOOK-GRAB	4-1-1	0.461	0.971	1.16	0.318
LOOK-GRAB	4-1-2	0.48	0.558	0.658	0.615
LOOK-GRAB	4-1-3	0.467	0.538	0.704	0.61
LOOK-GRAB	4-2-1	-	-	-	-
LOOK-GRAB	4-2-2	2.14	1.96	1.69	0.679
LOOK-GRAB	4-2-3	2.11	2.09	1.71	0.682
LOOK-GRAB	4-3-1	-	-	-	-
LOOK-GRAB	4-3-2	-	-	-	-
LOOK-GRAB	4-3-3	-	-	-	-
LOOK-GRAB	8-1-1	-	-	-	-
LOOK-GRAB	8-1-2	-	-	-	-
LOOK-GRAB	8-1-3	-	-	-	-
LOOK-GRAB	8-2-1	-	-	-	-
LOOK-GRAB	8-2-2	-	-	-	-
LOOK-GRAB	8-2-3	-	-	-	-
LOOK-GRAB	8-3-1	-	-	-	-
LOOK-GRAB	8-3-2	-	-	-	-
LOOK-GRAB	8-3-3	-	-	-	-
DISPOSE	2-2	1.66	1.75	1.45	0.321
DISPOSE	2-3	2.15	2.27	1.96	0.332
DISPOSE	3-2	2.61	2.4	2.25	0.225
DISPOSE	3-3	4.05	2.43	2.08	0/9
DISPOSE	4-2	4.37	2.19	3.58	0.301
DISPOSE	4-3	-	-	-	-

Continued on next page

Domain	Instance	Ratio-32	Ratio-16	Ratio-8	Ratio-1
DISPOSE	5-2	5/0	4/0	9/0	-
DISPOSE	5-3	-	-	-	-
DISPOSE	6-2	-	-	-	-
DISPOSE	6-3	-	-	-	-
UTS	1	0.237	0.333	0.333	0.45
UTS	2	0.543	0.543	0.625	0.357
UTS	3	0.894	0.778	0.857	0.636
UTS	4	0.922	0.845	0.91	0.25
UTS	5	1.22	0.538	0.621	0.31
UTS	6	0.321	0.718	0.844	0.382
UTS	7	0.488	0.824	0.809	0.275
UTS	8	0.685	0.549	0.845	0.547
UTS	9	2/9	0.502	1.14	0.438
UTS	20	0/9	3/9	0.941	0.394
UTS	30	0/9	0.448	0.573	0.422
UTS	40	0/9	0.449	0.476	0.278
UTS	50	-	-	-	-
UTS	60	-	-	-	-

Table 8: Speed-up factors of `parallel-CPCES` over `FF-Hit-Random` at $\tau = 0.90$. Each ratio-x column represents the speed-up achieved by `parallel-CPCES` with x workers relative to `FF-Hit-Random`.

Domain	Instance	Ratio-32	Ratio-16	Ratio-8	Ratio-1
BOMB	20-1	23.9	16.1	18.4	2.14
BOMB	20-5	20.1	13.6	21.6	3.69
BOMB	20-10	8.28	15.3	12.5	2.56
BOMB	20-20	-	-	-	-
BOMB	100-1	9/0	9/0	9/0	9/0
BOMB	100-5	9/0	9/0	9/0	9/0
BOMB	100-10	9/0	9/0	9/0	9/0
BOMB	100-60	9/0	9/0	9/0	7/0
BOMB	100-100	9/0	9/0	9/0	9/0
COINS	10	0.778	0.587	0.873	0.439
COINS	12	1.19	6.97	3.53	1.82
COINS	15	26.1	29.6	28.9	4.92
COINS	16	5.96	5.3	4.34	2.45
COINS	17	12.3	4.36	4.7	2.07
COINS	18	5.73	5.57	3.01	2.1
COINS	19	2.63	4.42	3.53	2.09
COINS	20	8.55	3.72	2.39	2.29
COINS	21	-	-	-	-
DISPOSE	4-1	4.86	3.14	7.95	1.44
DISPOSE	4-2	19.7	28.4	19.3	6.83
DISPOSE	4-3	26.2	18.9	17.1	3.09
DISPOSE	8-1	52.8	51.5	55.8	17.4
DISPOSE	8-2	9/0	9/0	9/0	9/0
DISPOSE	8-3	9/0	9/0	9/0	9/0
DISPOSE	12-1	9/0	9/0	9/0	9/0
DISPOSE	12-2	-	-	-	-

Continued on next page

Domain	Instance	Ratio-32	Ratio-16	Ratio-8	Ratio-1
DISPOSE	12-3	-	-	-	-
LOGISTICS	1	-	-	-	-
LOGISTICS	2	-	-	-	-
LOGISTICS	3	-	-	-	-
LOGISTICS	4	-	-	-	-
LOOK-GRAB	4-1-1	1.01	1.25	1.33	0.485
LOOK-GRAB	4-1-2	0.481	0.593	0.708	0.614
LOOK-GRAB	4-1-3	0.445	0.582	0.726	0.631
LOOK-GRAB	4-2-1	-	-	-	-
LOOK-GRAB	4-2-2	2.03	2.06	1.89	0.765
LOOK-GRAB	4-2-3	2.01	2.03	1.79	0.769
LOOK-GRAB	4-3-1	-	-	-	-
LOOK-GRAB	4-3-2	-	-	-	-
LOOK-GRAB	4-3-3	-	-	-	-
LOOK-GRAB	8-1-1	-	-	-	-
LOOK-GRAB	8-1-2	-	-	-	-
LOOK-GRAB	8-1-3	-	-	-	-
LOOK-GRAB	8-2-1	-	-	-	-
LOOK-GRAB	8-2-2	-	-	-	-
LOOK-GRAB	8-2-3	-	-	-	-
LOOK-GRAB	8-3-1	-	-	-	-
LOOK-GRAB	8-3-2	-	-	-	-
LOOK-GRAB	8-3-3	-	-	-	-
ONEDISPOSE	2-2	1.36	1.53	1.61	0.5
ONEDISPOSE	2-3	5.36	4.99	3.95	0.718
ONEDISPOSE	3-2	17.1	15.2	12.9	0.796
ONEDISPOSE	3-3	9/0	9/0	9/0	-
ONEDISPOSE	4-2	14.8	13.6	12.4	0.388
ONEDISPOSE	4-3	-	-	-	-

Continued on next page

Domain	Instance	Ratio-32	Ratio-16	Ratio-8	Ratio-1
ONEDISPOSE	5-2	6/0	6/0	5/0	-
ONEDISPOSE	5-3	-	-	-	-
ONEDISPOSE	6-2	-	-	-	-
ONEDISPOSE	6-3	-	-	-	-
UTS	1	0.455	0.476	0.5	0.526
UTS	2	0.558	0.545	0.585	0.407
UTS	3	0.979	0.723	0.77	0.662
UTS	4	0.174	0.971	0.932	0.739
UTS	5	0.248	0.562	0.485	0.515
UTS	6	3.89	3.96	3.96	1.71
UTS	7	3.2	3.77	4.14	1.15
UTS	8	0.948	3.64	3.86	1.53
UTS	9	2.63	5.84	4.96	2.13
UTS	20	4/9	7.06	5.87	3.3
UTS	30	2.08	3.15	3.44	1.81
UTS	40	1.21	1.43	1.77	1.04
UTS	50	-	-	-	-
UTS	60	-	-	-	-

Table 9: Speed-up factors of `parallel-CPCES` over `FF-Hit-Random` at $\tau = 0.75$. Each ratio-x column represents the speed-up achieved by `parallel-CPCES` with x workers relative to `FF-Hit-Random`.

Domain	Instance	Ratio-32	Ratio-16	Ratio-8	Ratio-1
BOMB	20-1	11.7	11.6	10.5	7.49
BOMB	20-5	13.3	13.4	8.62	3.01
BOMB	20-10	11.6	13.3	13.1	7.26
BOMB	20-20	-	-	-	-
BOMB	100-1	9/0	9/0	9/0	9/0
BOMB	100-5	9/0	9/0	9/0	9/0
BOMB	100-10	9/0	9/0	9/0	9/0
BOMB	100-60	9/0	9/0	9/0	9/0
BOMB	100-100	9/0	9/0	9/0	9/0
COINS	10	7.17	5.5	4.88	1.88
COINS	12	30.7	27.8	21.5	6.65
COINS	15	93.4	88.2	56.8	9.4
COINS	16	40.8	42.1	33.3	18.4
COINS	17	41.7	41.9	38	18.4
COINS	18	46.6	49.3	42.7	19.9
COINS	19	34.1	39.5	35.3	15.8
COINS	20	42	40.2	38	15.1
COINS	21	-	-	-	-
DISPOSE	4-1	10.5	11.7	11	5.18
DISPOSE	4-2	32.9	22.9	20.2	9.05
DISPOSE	4-3	33.2	25.5	26.3	9.02
DISPOSE	8-1	37.4	36.6	35.8	21.9
DISPOSE	8-2	9/0	9/0	9/0	9/0
DISPOSE	8-3	9/0	9/0	9/0	6/0
DISPOSE	12-1	9/0	9/0	9/0	9/0
DISPOSE	12-2	-	-	-	-

Continued on next page

Domain	Instance	Ratio-32	Ratio-16	Ratio-8	Ratio-1
DISPOSE	12-3	-	-	-	-
LOGISTICS	1	-	-	-	-
LOGISTICS	2	-	-	-	-
LOGISTICS	3	-	-	-	-
LOGISTICS	4	-	-	-	-
LOOK-GRAB	4-1-1	0.791	0.909	0.932	0.853
LOOK-GRAB	4-1-2	0.516	0.685	0.768	0.7
LOOK-GRAB	4-1-3	0.5	0.65	0.783	0.707
LOOK-GRAB	4-2-1	-	-	-	-
LOOK-GRAB	4-2-2	1.75	1.67	1.67	0.988
LOOK-GRAB	4-2-3	1.74	1.77	1.67	0.985
LOOK-GRAB	4-3-1	-	-	-	-
LOOK-GRAB	4-3-2	-	-	-	-
LOOK-GRAB	4-3-3	-	-	-	-
LOOK-GRAB	8-1-1	-	-	-	-
LOOK-GRAB	8-1-2	-	-	-	-
LOOK-GRAB	8-1-3	-	-	-	-
LOOK-GRAB	8-2-1	-	-	-	-
LOOK-GRAB	8-2-2	-	-	-	-
LOOK-GRAB	8-2-3	-	-	-	-
LOOK-GRAB	8-3-1	-	-	-	-
LOOK-GRAB	8-3-2	-	-	-	-
LOOK-GRAB	8-3-3	-	-	-	-
ONEDISPOSE	2-2	3.28	3.51	3.17	2.22
ONEDISPOSE	2-3	6.46	6.11	5.9	0.993
ONEDISPOSE	3-2	26.1	25.6	23.1	1.33
ONEDISPOSE	3-3	9/0	9/0	9/0	-
ONEDISPOSE	4-2	30.6	30.7	19.7	0.903
ONEDISPOSE	4-3	-	-	-	-

Continued on next page

Domain	Instance	Ratio-32	Ratio-16	Ratio-8	Ratio-1
ONEDISPOSE	5-2	8/0	6/0	6/0	-
ONEDISPOSE	5-3	-	-	-	-
ONEDISPOSE	6-2	-	-	-	-
ONEDISPOSE	6-3	-	-	-	-
UTS	1	0.333	0.36	0.346	0.333
UTS	2	0.5	0.542	0.553	0.5
UTS	3	1.67	1.55	0.828	0.837
UTS	4	6.24	6.7	5.66	2.46
UTS	5	15	16.3	16.6	11.9
UTS	6	20.4	20.9	20.8	10.9
UTS	7	1.48	5.1	5.38	3.03
UTS	8	4.67	5.32	5.74	2.35
UTS	9	3.36	3.73	3.89	2.77
UTS	20	5.27	5.82	6.2	4.16
UTS	30	2.47	2.63	2.68	1.78
UTS	40	1.78	1.84	1.86	1.31
UTS	50	0.954	0.969	0.973	0.381
UTS	60	-	-	-	-

Table 10: Speed-up factors of `parallel-CPCES` over `FF-Hit-Random` at $\tau = 0.50$. Each ratio-x column represents the speed-up achieved by `parallel-CPCES` with x workers relative to `FF-Hit-Random`.

Domain	Instance	Ratio-32	Ratio-16	Ratio-8	Ratio-1
BOMB	20-1	1.54	1.67	1.63	1.18
BOMB	20-5	1.59	1.4	1.51	1.17
BOMB	20-10	1.37	1.49	1.44	1.08
BOMB	20-20	1.26	1.4	1.45	1.14
BOMB	100-1	1.71	1.56	1.66	1.32
BOMB	100-5	1.53	1.51	1.7	0.684
BOMB	100-10	1.52	1.39	1.32	1.27
BOMB	100-60	1.35	1.49	1.49	1.35
BOMB	100-100	1.5	1.69	1.59	0.8
COINS	10	5.84	6.98	6.68	2.51
COINS	12	24.2	13.6	18.1	3.62
COINS	15	17.7	8.21	14.2	0.918
COINS	16	20.8	22.7	21.4	4.28
COINS	17	33.3	27.4	19.9	7.35
COINS	18	37.5	27	19.6	5.26
COINS	19	28.3	36.5	17	3.76
COINS	20	25.4	22.6	23.5	4.21
COINS	21	-	-	-	-
DISPOSE	4-1	55.5	51.5	52	22.3
DISPOSE	4-2	17.3	14.9	16.3	1.66
DISPOSE	4-3	26.4	26.5	20.3	5.75
DISPOSE	8-1	27.1	25.3	25.4	5.01
DISPOSE	8-2	9/0	9/0	9/0	9/0
DISPOSE	8-3	-	5/0	6/0	-
DISPOSE	12-1	9/0	9/0	9/0	9/0
DISPOSE	12-2	-	-	-	-

Continued on next page

Domain	Instance	Ratio-32	Ratio-16	Ratio-8	Ratio-1
DISPOSE	12-3	-	-	-	-
LOGISTICS	1	-	-	-	-
LOGISTICS	2	-	-	-	-
LOGISTICS	3	0.978	0.978	0.981	1.04
LOGISTICS	4	-	-	-	-
LOOK-GRAB	4-1-1	0.872	1.01	1.05	0.99
LOOK-GRAB	4-1-2	0.621	0.745	0.891	0.812
LOOK-GRAB	4-1-3	0.587	0.724	0.875	0.824
LOOK-GRAB	4-2-1	-	-	-	-
LOOK-GRAB	4-2-2	1.45	1.44	1.41	1.13
LOOK-GRAB	4-2-3	1.43	1.44	1.4	1.12
LOOK-GRAB	4-3-1	-	-	-	-
LOOK-GRAB	4-3-2	-	-	-	-
LOOK-GRAB	4-3-3	-	-	-	-
LOOK-GRAB	8-1-1	-	-	-	-
LOOK-GRAB	8-1-2	-	-	-	-
LOOK-GRAB	8-1-3	-	-	-	-
LOOK-GRAB	8-2-1	-	-	-	-
LOOK-GRAB	8-2-2	-	-	-	-
LOOK-GRAB	8-2-3	-	-	-	-
LOOK-GRAB	8-3-1	-	-	-	-
LOOK-GRAB	8-3-2	-	-	-	-
LOOK-GRAB	8-3-3	-	-	-	-
ONEDISPOSE	2-2	2.36	2.48	2.96	1.68
ONEDISPOSE	2-3	7.46	7.18	7.02	3.13
ONEDISPOSE	3-2	38	36.6	34.3	1.54
ONEDISPOSE	3-3	9/0	9/0	9/0	-
ONEDISPOSE	4-2	9/0	9/0	9/0	-
ONEDISPOSE	4-3	-	-	-	-

Continued on next page

Domain	Instance	Ratio-32	Ratio-16	Ratio-8	Ratio-1
ONEDISPOSE	5-2	8/0	5/0	6/0	-
ONEDISPOSE	5-3	-	-	-	-
ONEDISPOSE	6-2	-	-	-	-
ONEDISPOSE	6-3	-	-	-	-
UTS	1	0.4	0.414	0.429	0.545
UTS	2	0.561	0.575	0.605	0.575
UTS	3	1.06	1.02	1.06	0.981
UTS	4	3.19	3.25	3.14	1.1
UTS	5	11.1	13.1	13.6	10.5
UTS	6	45.7	50.8	42.2	25.2
UTS	7	153	205	181	160
UTS	8	27.4	31.2	32.7	19.3
UTS	9	18.9	21.3	22.1	9.41
UTS	20	15.4	17.1	17.8	8.24
UTS	30	5.95	6.36	6.51	4.82
UTS	40	3.56	3.7	3.72	2.51
UTS	50	1.5	1.53	1.54	0.653
UTS	60	-	-	-	-

Bibliography

Armando, A. and Compagna, L. (2008). SAT-based model-checking for security protocols analysis. *International Journal of Information Security*, 7:3–32.

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

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

Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. (2001). Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 203–213.

Basin, D., Cremers, C., and Meadows, C. (2018). Model checking security protocols. *Handbook of Model Checking*, pages 727–762.

Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., and Schnoebelen, P. (2013). *Systems and software verification: Model-checking techniques and tools*. Springer Science & Business Media.

Bertoli, P., Cimatti, A., Roveri, M., et al. (2001). Heuristic search + symbolic model checking = efficient conformant planning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, volume 1, pages 467–472. Citeseer.

Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., and Pollitt, F. (2024). CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In Heule, M., Iser, M., Järvisalo, M., and Suda, M., editors, *Proceedings of SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions*,

volume B-2024-1 of *Department of Computer Science Report Series B*, pages 8–10. University of Helsinki.

Biere, A., Heule, M., and van Maaren, H. (2009). *Handbook of satisfiability*, volume 185. IOS press.

Bläsius, T., Friedrich, T., Stangl, D., and Weyand, C. (2022). An efficient branch-and-bound solver for hitting set*. In *Proceedings of the 2022 Symposium on Algorithm Engineering and Experiments*, pages 209–220. SIAM.

Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300.

Bonet, B. (2010). Conformant plans and beyond: Principles and complexity. *Artificial Intelligence*, 174(3-4):245–269.

Bonet, B. and Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, pages 52–61.

Bonet, B. and Geffner, H. (2014). Belief tracking for planning with sensing: Width, complexity and approximations. *Journal of Artificial Intelligence Research*, 50:923–970.

Bonet, B., Loerincs, G., and Geffner, H. (1997). A robust and fast action selection mechanism for planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and the Ninth Innovative Applications of Artificial Intelligence Conference*, pages 714–719.

Bonet, B., Palacios, H., and Geffner, H. (2009). Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, volume 19, pages 34–41.

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

Bryce, D., Kambhampati, S., and Smith, D. E. (2006a). Planning graph heuristics for belief space search. *Journal of Artificial Intelligence Research*, 26:35–99.

Bryce, D., Kambhampati, S., and Smith, D. E. (2006b). Sequential monte carlo in probabilistic planning reachability heuristics. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, pages 233–242.

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

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

Cimatti, A. and Roveri, M. (2000). Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*, 13:305–338.

Cimatti, A., Roveri, M., and Bertoli, P. (2004). Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 159(1-2):127–206.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In Emerson, E. A. and Sistla, A. P., editors, *Proceedings of the Twelfth International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, Chicago, IL, USA. Springer.

Clarke, E. M. (1997). Model checking. In Ramesh, S. and Sivakumar, G., editors, *Proceedings of the Seventeenth Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *Lecture Notes in Computer Science*, pages 54–56, Kharagpur, India. Springer.

Clarke, E. M., Emerson, E. A., and Sifakis, J. (2009). Model checking: Algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84.

Clarke, E. M., Grumberg, O., and Long, D. E. (1994). Model checking and abstraction. *ACM transactions on Programming Languages and Systems*, 16(5):1512–1542.

Darwiche, A. (2001). On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34.

Darwiche, A. and Marquis, P. (2002a). A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264.

Darwiche, A. and Marquis, P. (2002b). A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264.

Dechter, R. (2003). *Constraint processing*. Elsevier.

Derrick, J. and Wehrheim, H. (2007). On using data abstractions for model checking refinements. *Acta Informatica*, 44:41–71.

Domshlak, C. and Hoffmann, J. (2006). Fast probabilistic planning through weighted model counting. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, pages 243–252.

Domshlak, C. and Hoffmann, J. (2007). Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research*, 30:565–620.

Doucet, A., de Freitas, N., and Gordon, N. (2001). *Sequential Monte Carlo methods in practice*. Springer.

Edelkamp, S. and Hoffmann, J. (2004). PDDL2.2: The language for the classical part of the fourth international planning competition. Technical Report Technical Report 195, Institut für Informatik, Albert-Ludwigs-Universität Freiburg.

Edwards, S. A., Ma, T., and Damiano, R. (2001). Using a hardware model checker to verify software. In *Proceedings of the Fourth International Conference on ASIC*, pages 85–90. IEEE.

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

Finkbeiner, B., Rabe, M. N., and Sánchez, C. (2015). Algorithms for model checking HyperLTL and HyperCTL. In *Proceedings of the Twenty-Seventh In-*

ternational Conference on Computer Aided Verification, Part I, pages 30–48. Springer.

Fox, M. and Long, D. (2003). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124.

Gainer-Dewar, A. and Vera-Licona, P. (2017). The minimal hitting set generation problem: Algorithms and computation. *SIAM Journal on Discrete Mathematics*, 31(1):63–100.

Garey, M. R. and Johnson, D. S. (2002). *Computers and intractability*, volume 29. W. H. Freeman, New York.

Geffner, H. (2000). Functional STRIPS: A more flexible language for planning and problem solving. In *Logic-Based Artificial Intelligence*, pages 187–209. Springer.

Geffner, H. (2007). The causal graph heuristic is the additive heuristic plus context. In *Proceedings of the Workshops at the Seventeenth International Conference on Automated Planning and Scheduling*.

Gerevini, A. and Long, D. (2005). Plan constraints and preferences in PDDL3. Technical Report Technical Report 2005-08-07, Department of Electronics for Automation, University of Brescia, Italy.

Grastien, A. and Scala, E. (2020). CPCES: A planning framework to solve conformant planning problems through a counterexample guided refinement. *Artificial Intelligence*, 284:103271.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

Haslum, P. and Jonsson, P. (2000). Some results on the complexity of planning with incomplete information. In Biundo, S. and Fox, M., editors, *Proceedings of the Fifth European Conference on Planning*, volume 1809 of *Lecture Notes in Computer Science*, pages 308–318, Durham, UK. Springer.

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

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

Hoffmann, J. (2001). FF: The fast-forward planning system. *AI Magazine*, 22(3):57–57.

Hoffmann, J. and Brafman, R. I. (2006). Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, 170(6-7):507–541.

Holzmann, G. J. and Joshi, R. (2004). Model-driven software verification. In *Proceedings of the Eleventh International SPIN Workshop on Model Checking of Software*, volume 2989, pages 76–91. Springer.

Huang, J. (2006). Combining knowledge compilation and search for conformant probabilistic planning. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, pages 253–262.

Huang, J. and Darwiche, A. (2007). The language of search. *Journal of Artificial Intelligence Research*, 29:191–219.

Hyafil, N. and Bacchus, F. (2003). Conformant probabilistic planning via CSPs. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling*, volume 98, pages 205–214.

Hyafil, N. and Bacchus, F. (2004). Utilizing structured representations and CSPs in conformant probabilistic planning. In *Proceedings of the 16th European Conference on Artificial Intelligence*, volume 16, page 1033.

Ignatiev, A., Morgado, A., and Marques-Silva, J. (2018). RC2: A python-based MaxSAT solver. *MaxSAT Evaluation*, 2018:22.

Jhala, R. and Majumdar, R. (2009). Software model checking. *ACM Computing Surveys*, 41(4):1–54.

Karp, R. M. (2009). Reducibility among combinatorial problems. In *50 Years of*

Integer Programming 1958-2008: from the Early Years to the State-of-the-Art, pages 219–241. Springer.

Kautz, H. and Selman, B. (1999). Unifying SAT-based and graph-based planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, volume 99, pages 318–325.

Kavvadias, D. and Stavropoulos, E. (2005). An efficient algorithm for the transversal hypergraph generation. *Journal of Graph Algorithms and Applications*, 9(2):239–264.

Kurien, J., Nayak, P. P., and Smith, D. E. (2002). Fragment-based conformant planning. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling*, pages 153–162. AAAI Press.

Kushmerick, N., Hanks, S., and Weld, D. S. (1995). An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1-2):239–286.

Li, Y. (2021). Multi-agent conformant planning with distributed knowledge. In *Logic, Rationality, and Interaction: Eighth International Workshop*, pages 128–140. Springer.

Lowe, G. (1999). Towards a completeness result for model checking of security protocols. *Journal of computer security*, 7(2-3):89–146.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D., Barrett, A., Christianson, D., Etzioni, O., Pednault, E., et al. (1998). PDDL: The Planning Domain Definition Language. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control.

Muise, C., McIlraith, S. A., Beck, J. C., and Hsu, E. I. (2012). DSHARP: Fast d-DNNF compilation with sharpSAT. In *Proceedings of the Twenty-Fifth Canadian Conference on Artificial Intelligence*, volume 7310, pages 356–361. Springer.

Newell, A. and Simon, H. A. (1961). GPS, a program that simulates human thought. Technical Report P-2257, The RAND Corporation, Santa Monica, California.

Nguyen, K., Tran, V., Son, T., and Pontelli, E. (2012). On computing conformant plans using classical planners: A generate-and-complete approach. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, volume 22, pages 190–198.

Onder, N., Whelan, G. C., and Li, L. (2006). Engineering a conformant probabilistic planner. *Journal of Artificial Intelligence Research*, 25:1–15.

Palacios, H. and Geffner, H. (2009). Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research*, 35:623–675.

Pearl, J. (1984). *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc.

Pohl, I. (1970). Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3-4):193–204.

Pryor, L. and Collins, G. (1996). Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339.

Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95.

Richter, S., Helmert, M., and Westphal, M. (2008). Landmarks revisited. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, volume 8, pages 975–982.

Richter, S. and Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177.

Richter, S., Westphal, M., and Helmert, M. (2011). LAMA 2008 and 2011. In *Proceedings of the Seventh International Planning Competition, Deterministic Part*, pages 50–54.

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

Rintanen, J. (2014). Madagascar: Scalable planning with SAT. In *Proceedings of the Eighth International Planning Competition: Description of Participating Planners, Deterministic Track*, pages 66–69.

Rintanen, J., Heljanko, K., and Niemelä, I. (2004). Parallel encodings of classical planning as satisfiability. In *Proceedings of the Ninth European Conference on Logics in Artificial Intelligence*, pages 307–319. Springer.

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.

Russell, S. J. and Norvig, P. (2020). *Artificial intelligence: A modern approach*. Pearson, United States, 4 edition.

Seipp, J. and Helmert, M. (2018). Counterexample-guided cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, 62:535–577.

Slaney, J. K. (2014). Set-theoretic duality: A fundamental feature of combinatorial optimisation. In *Proceedings of the Twenty-First European Conference on Artificial Intelligence*, pages 843–848.

Smith, D. E. and Weld, D. S. (1998). Conformant graphplan. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 889–896.

Taig, R. and Brafman, R. I. (2012). Using classical planners to solve conformant probabilistic planning problems. In *Proceedings of the AAAI-12 Workshop on Heuristics for Domain-Independent Planning*. AAAI Press.

Taig, R. and Brafman, R. I. (2013). Compiling conformant probabilistic planning problems into classical planning. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, volume 23, pages 197–205.

To, S., Son, T., and Pontelli, E. (2010). A new approach to conformant planning using CNF. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, volume 20, pages 169–176.

Winston, P. H. (1992). *Artificial intelligence*. Addison-Wesley Longman Publishing Co., Inc.

Younes, H. L. and Littman, M. L. (2004). PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. Technical Report CMU-CS-04-162, Computer Science Department, Carnegie Mellon University.

Younes, H. L. and Simmons, R. G. (2003). VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, 20:405–430.

Zhang, X. and Grastien, A. (2023). Improvements to CPCES. In *Proceedings of the Sixteenth International Symposium on Combinatorial Search*, volume 16, pages 110–118.

Zhang, X., Grastien, A., and Scala, E. (2020). Computing superior counterexamples for conformant planning. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence*, pages 10017–10024. AAAI Press.