Author/s:
Singh, Anubhav

Title:
Lazy Constraint Generation and Tractable Approximations for Large Scale Planning Problems

Date:
2023-12

Persistent Link:
https://hdl.handle.net/11343/344694

Lazy Constraint Generation and Tractable Approximations for Large-scale Planning Problems

by

Anubhav Singh

ORCID: 0000-0003-2770-801X

Submitted in total fulfillment for the
degree of Doctor of Philosophy

in the
School of Computing and Information Systems
The University of Melbourne
THE UNIVERSITY OF MELBOURNE

April 2024

Abstract

In our research, we explore two orthogonal but related methodologies of solving planning instances: planning algorithms based on direct but lazy, incremental heuristic search over transition systems and planning as satisfiability. We address numerous challenges associated with solving large planning instances within practical time and memory constraints. This is particularly relevant when solving real-world problems, which often have numeric domains and resources and, therefore, have a large *ground* representation of the planning instance. Our first contribution is an *approximate novelty search*, which introduces two novel methods. The first approximates *novelty* via sampling and Bloom filters, and the other approximates the best-first search using an adaptive policy that decides whether to forgo the expansion of nodes in the open list. For our second work, we present an encoding of the partial order causal link (POCL) formulation of the temporal planning problems into a CP model that handles the instances with *required concurrency*, which cannot be solved using sequential planners. Our third significant contribution is on *lifted sequential planning with lazy constraint generation*, which scales very well on large instances with numeric domains and resources. Lastly, we propose a novel way of using novelty approximation as a polynomial reachability propagator, which we use to train the activity heuristics used by the CP solvers.

Declaration of Authorship

I, Anubhav Singh, declare that this thesis titled, 'Lazy Constraint Generation and Tractable Approximations for Large Scale Planning Problems' and the work presented in it are my own. I confirm that:

- The thesis comprises only my original work towards the Doctor of Philosophy except where indicated in the preface;
- due acknowledgement has been made in the text to all other material used; and
- the thesis is fewer than the maximum word limit in length, exclusive of tables, maps, bibliographies and appendices as approved by the Research Higher Degrees Committee.

Signed: Anubhav Singh

Date: 07 April 2024

Preface

This dissertation contains original work conducted in the Computing and Information Systems department at the University of Melbourne under the supervision of Dr. Nir Lipovetzky, Dr. Miquel Ramirez, and Prof. Peter Stuckey.

Chapter 4 of the thesis includes the following published papers

- Anubhav Singh, Nir Lipovetzky, Miquel Ramirez, and Javier Segovia-Aguas. "Approximate novelty search." In *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 31, pp. 349-357. 2021.
- Anubhav Singh, Nir Lipovetzky, Miquel Ramirez and Javier Segovia-Aguas. "Approximate Novelty Search." In *Int. Planning Competition Booklet (IPC-10)*, 2023
- Anubhav Singh, Nir Lipovetzky, Miquel Ramirez, Javier Segovia-Aguas, Guillem Frances. "Grounding Schematic Representation with GRINGO for Width-based Search." In *Int. Planning Competition Booklet (IPC-10)*, 2023

The contents of Chapters 5, 6 and 7 are unpublished original works exploring the computational paradigm of *planning as constraint programming (CP)* which would be submitted to a Journal. Prof. Peter Stuckey from Monash University advised us on these works.

Acknowledgements

I am deeply thankful to my Ph.D. advisors for guiding me toward challenging and fascinating problems in AI planning and equally interesting fields of satisfiability and mathematical programming. It helped broaden my horizons and gain more profound, practical insights into knowledge representation and reasoning. Researching with them has fostered a curiosity for learning that I hope will persist throughout my life.

To Nir. I am grateful for the liberty you gave me to think and explore independently and for keeping me motivated when I doubted myself and when the accumulated emotional burden of social isolation during COVID-19 began to affect everything else. I enjoyed communicating freely and honestly and sharing my ideas with you. Thank you for inspiring me to follow this path.

To Miguel. I am very grateful for your guidance in aligning my ambitions and reality. Your help in grasping each new domain we investigated was indispensable, and I learned something new in every interaction with you. Thank you for enlightening me on the importance of robust theory and analysis and helping me understand the formalization underlying our research.

To Peter. I appreciate you listening to my ideas, correcting my mistakes, and helping me navigate the Constraint and Mathematical Programming fields. Conversations with you stimulated me to undertake more research and feel great about it.

To Lyndon and Marcus. Thank you both for those long conversations where we lost track of time, bouncing ideas off each other. They expanded my perspectives and clarified my doubts, helping me make the uncertain certain.

To the fellow members of the Artificial Intelligence Lab: Archie, Rinu, Chao, Steven, Stefan, Prashan, Guang, Ruihan, Victoria, Emma, Michelle, Tim, Vincent, Chen, Thao, Chenyuan, Abeer, and Hanan. Thank you for giving me a sense of community. I will always remain proud to be part of such a vibrant and diverse group of brilliant people.

To my parents and brother. I appreciate you always motivating me to excel and encouraging me to look forward to the future rather than dwell on the past. I am grateful for my mum's phone calls, as she tried to cheer me up, urging me to smile and be social and happy when I was overwhelmed and lost track of such things.

Finally, I thank my lifelong friends Prashasti, Nitish, and Agastya for the countless treasured memories.

Contents

List of Figures

List of Tables

"If you want to be creative, then you will have to get used to spending most of your time not being creative, to being becalmed on the ocean of scientific knowledge."

Steven Weinberg

1 Introduction

AI Planning is a branch of artificial intelligence that concerns the realization of strategies, typically for execution by intelligent agents like autonomous robots and unmanned vehicles, that are required to make independent decisions. AI Planning aims to solve a wide range of problems with limited assumptions on the system's dynamics, allowing for the general use of computational approaches to planning — independent of the problem domain. Over time, different classes of assumptions on problem structure have given rise to sub-fields in planning, from the declarative representations of deterministic transition systems in *classical* and *temporal planning*, describing the initial state, goal states, and the actions clearly, to Markov Decision Processes (MDPs) with probabilistic outcomes in the paradigm of planning under uncertainty. Communities in each of these fields develop and employ specialized computational approaches. However, common touchpoints exist, and the standard methods are typically generalizable within the dynamic programming framework. Some planning formalisms also allow us to optimize the overall performance measures of the system, such as time, cost, and quality.

While the models of AI Planning contain little to no explicit information about the solutions, they do describe the dynamics of the world, *differentiating* planning approaches from *reinforcement learning* for model-free problems. However, much work has been done in connecting research in planning on MDPs and reinforcement learning, leading to model-based reinforcement learning. This has allowed the community to apply AI Planning approaches in many artificial problems, including single and multiplayer games with varying degrees of observability and explicit knowledge of the transition system, like Pacman, Checkers, Go, and Atari 2600 video games, as well as real-world problems of logistics, motion and task planning, and coordination in a multi-agent setting.

Although planning approaches have achieved remarkable results in various problems, there are still many challenges that limit their applicability and performance. One of the main challenges is scalability, which refers to the ability of planning systems to handle large and complex problems efficiently. Another challenge is the ease of real-world integration, which refers to the effort required to model the dynamics of real-world problems in planning systems' language. These challenges persist even when the world is modeled in a simplified and deterministic way, as in classical and temporal planning, where actions have fixed preconditions and effects, and the state of the world is fully known. Furthermore, the research in the computational approaches to planning seems to have reached a stagnation point, as evidenced by the last two International Planning Competitions, where the same technologies or portfolios of dominant methods have maintained their superiority over the others, with the framework of heuristic-driven forward search, that uses an estimator of the cost or the quality of a partial plan to guide the search for a complete plan, being the common denominator.

In this research, we seek to push the boundaries of generating agent behavior for general problem-solving as planning, focusing our attention on the two main classes of deterministic planning problems: sequential and temporally expressive. We explore two different computational paradigms of planning as heuristic search and planning as constraint programming as a basis for developing methods to tackle these challenges, where we aim to understand and address the inherent limitations of the state-of-the-art heuristic search methods in the former and develop encodings and planning specific decompositions in the latter. We explore thoroughly the techniques and progress in various areas of computer science and operations research and examine how they could enhance the performance of

planning, including computational methods in *heuristic search*, *stream computing*, *linear and mixed-integer programming*, *satisfiability*, and *constraint programming*.

1.1 Research Challenges

In this section, we describe the difficulties that arise in large and complex planning domains, and then we elaborate on the specific research questions we address.

1.1.1 Large-scale Problems in Planning

Planning problems involving large-scale scenarios and situations can present themselves in various forms and sizes, depending on the domain and the context. These planning problems also have specific attributes and features that make them more challenging to solve, stressing planning algorithms and search strategies designed to find optimal or satisficing solutions. Therefore, it is essential to understand their nature and characteristics and to evaluate the performance and suitability of different computational approaches for solving them.

Hard-to-ground. Consider Letter Boxed, a popular word game that is featured in the New York Times [43]. It challenges the players to find a sequence of words that connects the vertices on the edges of a square grid. This word game can be seen as a variation of the minimum edge cover problem, which is a well-known problem in graph theory and computer science. The edge cover problem asks for a minimum set of edges that touch all the vertices of a graph. Similarly, the Letter Boxed game asks for the shortest sequence of words that touch all the vertices on the edges of the square. The words that the players can use are constrained by the rules of the game, such as using only English words and following the order of the letters in the vertices on the grid. Letter Boxed is fun and challenging and also a surprisingly tricky problem for best-performing classical planning solvers from International Planning Competitions.

The Letter Boxed game requires the player to choose an initial vertex that fixes the first letter and the current edge, after which the player must connect it to a vertex on a different side from the current one. For example, in Figure 1.1, one could start with the vertex containing the letter 'C', connecting it to 'L' on the adjacent edge, and so on, until the word 'CLINIC' is composed. The player can make any sequence of vertices in

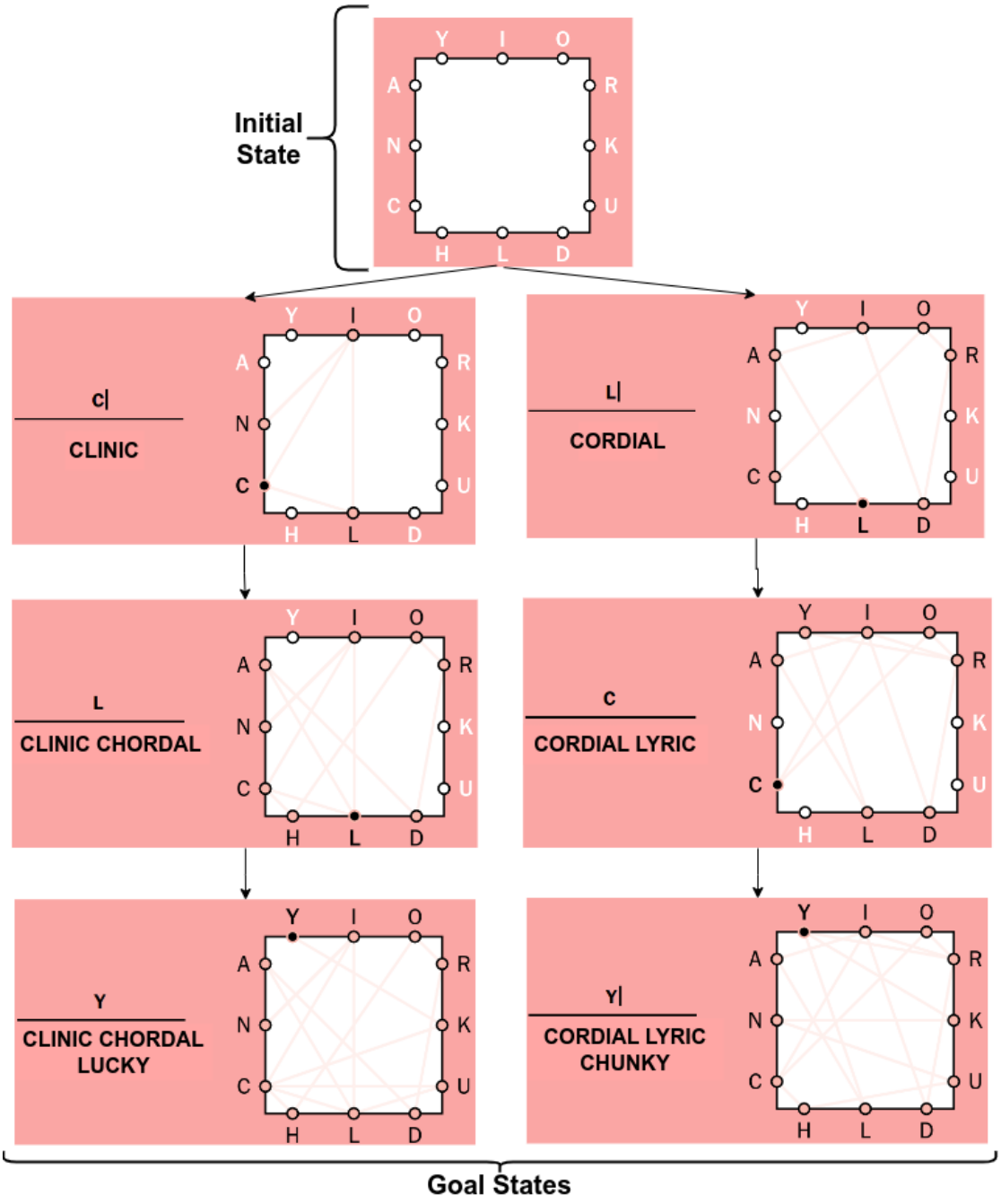


FIGURE 1.1: Diagramatic representation of the Letter Boxed[1] Transition System.

this manner. However, only a sequence comprising an English word is admissible. Once the user *enters* (finalizes) a word, the last letter of the previous word becomes the first letter of the next word; as we can see in the Figure, the next word, 'CHORDAL', starts with a 'C,' which is the last letter of 'CLINIC'. The player wins when all the vertices on the square have been visited.

[1] © New York Times Company

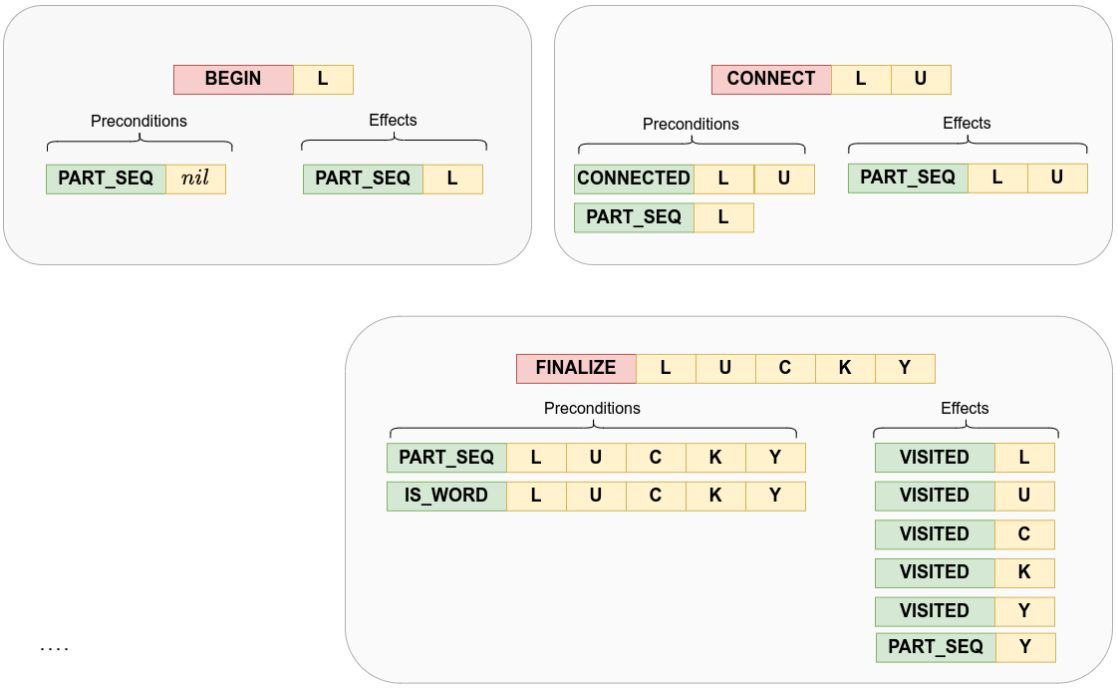


FIGURE 1.2: Diagramatic representation of the Letter Boxed Transition System.

The dynamics of the game can be easily captured in any standard input language for planning, for example, PDDL [60] with *begin*, *connect*, and *finalize* action schemas representing the act of selecting the first vertex, connecting vertices and entering a sequence to check admissibility, as shown in Figure 1.2. The game is simple for a human player with a good vocabulary. Still, it is challenging for planners as the knowledge of words is only encoded in the initial state, and the planner must use the dependency of action schemas on this knowledge base to infer the action arguments that constitute admissible words. This is considerably hard for *grounded* planners as the English letter words can be as long as 45 letters, requiring *connect* action schema with that many arguments, and a trivial *grounding* approach would result in an extremely large *ground* theory. Therefore, the Letter Boxed domain is *hard-to-ground* (HTG) for any planner based on planning approaches that require *grounding* lifted schematic representation of the problem domain. Furthermore, Letter Boxed isn't the only domain that stresses the *grounding* approaches, and many such problems have been captured in the recently released HTG benchmark [29, 89].

High Width. Many large-scale problems have attributes different from *hard-to-ground* that make them complex to solve. One such feature is *width*. Lipovetzky and Geffner [91, 94] proposed a measure of *width* of the problem along with a search strategy, *Iterated Width*, that guaranteed a runtime complexity exponential in the width of the problem.

They demonstrated that most benchmark instances appear to have low width when the goals were restricted to single atoms, resulting in low polynomial complexity. More advanced approaches of *Serialized* Iterated Width and Best First Width Search [93, 92] seek to exploit the width measure to decompose the problem and guide the search strategy, leading to state-of-the-art performance in the International Planning Competitions (IPCs). Still, many instances with *high width* remain hard to solve with width-based search as the computational complexity grows exponentially with the width of the problem.

Complex system of constraints. Logistics is one of the crucial problems, highly relevant from the applied standpoint, where the complex system of constraints makes the problem hard to solve. The problem involves moving packages from one location to another using various transportation methods. The problem is challenging because many factors and limitations affect the optimal solution. Figure 1.3 illustrates the problem with an example, where the packages 'Pi' have to be delivered to different destinations using airplanes and trucks. The problem combines three types of problems in operations research: *assignment* problem, *sequencing* problem, and *routing* problems, requiring the planner to find the best way to assign packages to transportation modes — which packages should go on which airplane or truck, ordering the package transportation, and deciding on the routes used to move around the packages to minimize the distance, time, or cost. The problem requires finding a solution that satisfies all these problems simultaneously while also considering the constraints and objectives of the logistics system.

1.1.2 Required Concurrency

While Sequential planning addresses the problems of assignment, sequencing, and routing, temporal planning adds the concurrency concern. Dealing with temporally expressive languages — durative action schemas with a structure of preconditions and effects that require *concurrent execution of two or more actions* to succeed, exploiting parallelism and managing dependencies among actions to optimize the plan metrics, typically *makespan*.

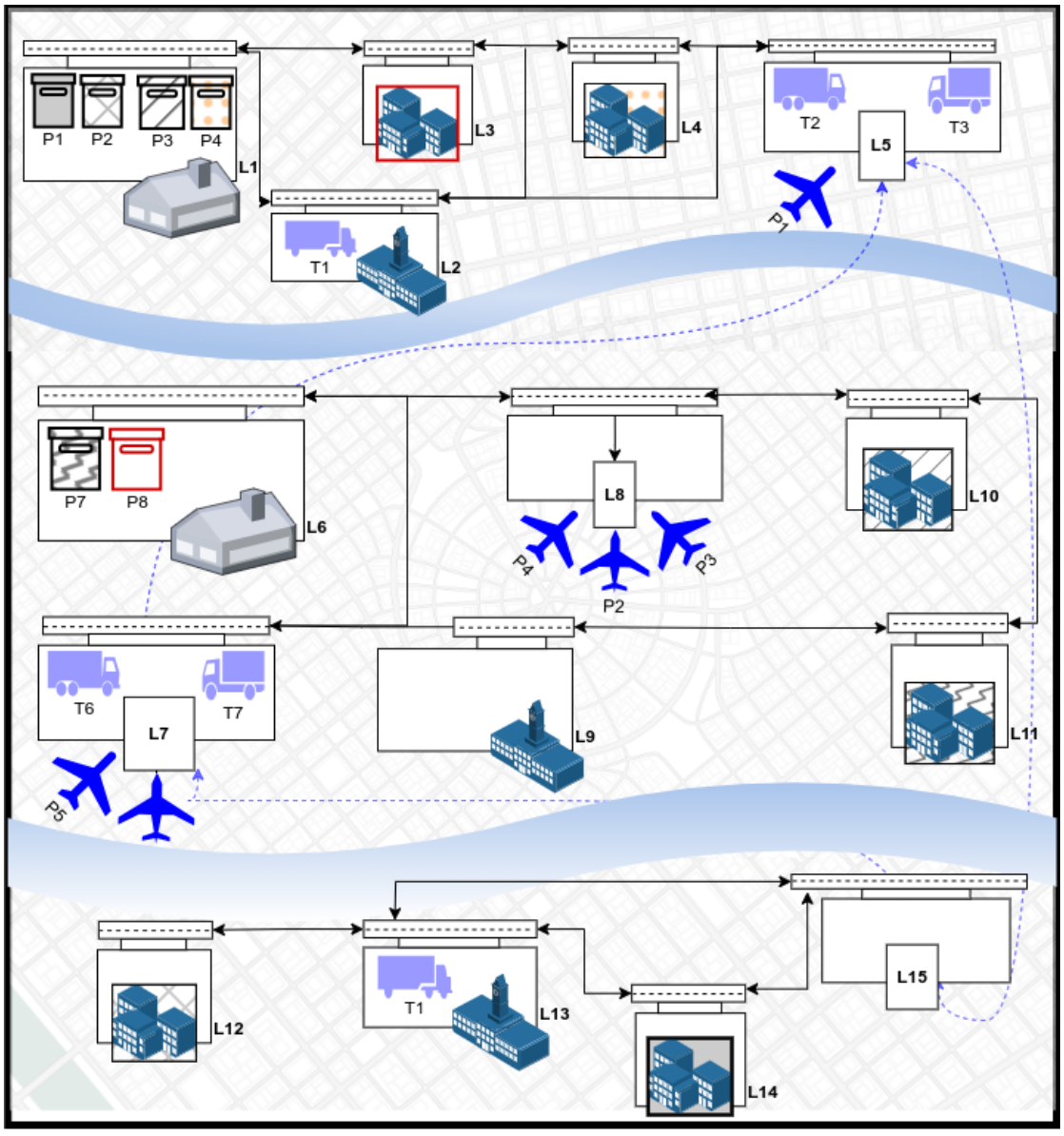


FIGURE 1.3: Diagramatic representation of a Logistics problem.

1.2 Research Goals

In this research, we aim to explore the different computational approaches for deterministic planning, focusing on understanding and adapting technological breakthroughs in operations research for planning. Furthermore, we aim to investigate decompositions of planning into operation research problems where efficient solving technology exists within the framework of Logic-based Benders Decomposition [75], addressing the challenges posed by large-scale planning problems. In addition to examining the methods in operations research, we seek to address immediate scalability concerns in planning as

width-based search — heuristic search framework synonymous with state-of-the-art in *satisficing* planning in the past decade.

Our **initial goal** is to design *approximations* of width-based planning algorithms [93, 92], addressing the scalability issue in *width-based search*, where the challenge is to reduce the space and time complexity of *novelty computation* to linear, enabling the planner to solve high width problem domains. Chakrabati et al. and Vadlamudi et al. [23, 146] seek to address the issue of scalability by trading off time for space, and Dionne et al. [34] trades-off optimality for assurance on time while preserving completeness. In this research, we would instead explore the idea of trading-off accuracy for time and memory guarantees using approximation methods. We hope to bring about a promising new direction with probabilistically complete search algorithms that can tackle the increasingly intractable benchmarks within hard constraints on time and memory.

We aim to *develop a scalable CP encoding of classical planning* as our **second research objective**, seeking to address the challenges posed by large-scale *hard-to-ground* planning instances. Our primary objective here is to tap into the groundbreaking technology of *Lazy Clause Generation* that allows us to represent a large number of constraints implicitly, considerably reducing the memory overhead. By developing scalable CP encoding of classical planning, we hope to advance the state-of-the-art in automated planning and constraint programming.

We further explore additional avenues in constraint programming, including planning-specific decompositions — breaking down a planning problem into smaller and simpler subproblems that can be solved more efficiently by constraint satisfaction approaches, designing and implementing custom search strategies that guide the constraint programming solver to find reasonable solutions quickly, and planning specific constraints and their implementations, propagators.

Lastly, we seek to address the concern of *concurrency* in temporal planning and develop an efficient planner that can be used to obtain optimal plans. For this, we will explore Constraint Programming and Mixed Integer Programming approaches to solving constraint satisfaction problems and develop encodings of temporal planning problems that solve temporally expressive domains — problems requiring *concurrent execution of actions* that cannot be solved with the common strategy of relaxing the temporal planning problem into a sequential planning one [57]. The previous work on the constraint-based

approach to temporal planning CPT [152] has directly inspired our research direction. We seek to address the shortcomings of CPT, including the inability to handle problems with *required concurrency* [31].

1.3 Contributions

Our research has resulted in four significant contributions that address the obstacles of large-scale planning problems.

1. In our first contribution, **Approximate novelty search**, we introduce novel polynomial approximations of state novelty and width-based search. It uses *Bloom filters* to *efficiently* represent the interpretation of the relational predicate and *random sampling* of tuples, conjunction of atoms, in the computation of state novelty. It also uses an adaptive policy which decides to delay the generation of successor states, based on the analytical solution to an MDP where its cost function determines the distribution of different novelty levels in the open list.

2. Our contribution on **Lifted Sequential Planning with Lazy Clause and Constraint generation solvers** presents a novel CP encoding that addresses the challenge of *hard-to-ground* (HTG) instances. We also demonstrate a planning-specific constraint implementation, propagator, in Google's CpSat solver that implements the Lazy Clause Generation approach in the CDCL framework, enabling the solver to scale better on HTG instances. We also introduce an automated method for *concise* conversion of planning problems described in PDDL to FSTRIPS [51] using the h^m heuristic over first-order logic existential formulae.

3. We also examine the two frameworks — the forward search paradigm and LCG CDCL — and suggest a combined method, **integrating planning as search and planning as CP**. Our research shows that blind search methods, such as breadth-first search and depth-first search, can be easily integrated into planning as Constraint Programming (CP). This integration is done by simple changes to the variable selection heuristic along with our encoding for Lifted Sequential Planning. Furthermore, we show that value selection heuristics in CDCL solvers can be used to select the successor for expansion, similar to planning heuristics. Finally, we define the concept of novelty as a constraint

within CP and present an effective implementation of novelty bound using a polynomial-time reachability propagator. The LCG CDCL-based algorithm, Depth First Novelty Search, using depth-first search heuristics and novelty propagator, leads to better results in our Lifted Encoding of Sequential Planning.

4. Lastly, we contribute a novel **CP encoding of optimal temporal POCL planning into CpSat that handles problems with *required concurrency***. The encoding is inspired by the CP encoding of the CPT planner which only handles simpler temporal planning problems, without required concurrency. We apply our CP encoding of POCL to the CpSat solver, and the resulting planner works very well in the instances with a high degree of concurrency, as well as in optimal temporal planning.

1.4 Thesis Outline

We start the thesis by reviewing computational approaches to classical planning and the recent advancements in the field in Chapter 2. In Chapter 3, we introduce the concern of concurrency along with the formalism of temporal planning. We then introduce our first contribution on Tractable approximations for width-based algorithms in classical planning in Chapter 4. We explain in formal detail the underlying scalability issues in width-based algorithms, including a measure of *novelty* whose complexity is exponential in the novelty value and the exponential growth of the open list with novelty bound. We prove the theoretical probability of error in novelty approximation and conclude by presenting the Apx Novelty Search planner with impressive performance on the IPC benchmark. In Chapter 5, we introduce challenges associated with planning in problems with large ground representation, which focuses on *hard-to-ground* instances. These instances feature domain theories of large size that render *grounding* the *lifted* representation into propositional STRIPS *intractable*. We present a Lifted Encoding of Sequential Planning in Constraint Programming and its implementation in Google's CpSat solver. We present a novel planning-specific user-propagator for *persistence constraints* that implicitly represents a large tract of complex constraints, allowing the solver to scale better. We show that the planner scales well on *hard-to-ground* benchmark while holding its ground on the IPC instances. We then discuss the integration of Planning as Heuristics Search and Planning as Constraint Programming in Chapter 6. We present an approach to implement forward search in planning into the framework of LCG CDCL

in CpSat. We also formalize novelty bound in width-based planning as a constraint and show an efficient implementation of Novelty propagator in CpSat. Lastly, in Chapter 7, we present an encoding of temporal planning in CP, which can handle required concurrency. We show that the encoding has state-of-the-art performance in instances with required concurrency. However, solving instances with a high degree of sequencing is a challenge.

Part I

Background

"Ramanujan was an artist. And numbers — and the mathematical language expressing their relationships — were his medium."

Robert Kanigel, The Man Who Knew Infinity : A Life of the Genius Ramanujan

2

Classical Planning

Classical Planning is a branch of artificial intelligence that deals with the problem of searching for behaviors in the form of an action sequence that maps the system's initial state into a desired goal situation. It provides a general and expressive framework for representing and solving many problems and sound mechanisms to reason over the space of plans. It has inspired other forms of planning, such as temporal, probabilistic, and hierarchical planning. This chapter explains the theory and practice of classical planning, focusing on the aspects of the classical planning model, description languages, theoretical properties, and computational methods. We also give an overview of the basic concepts from optimization theory and operations research relevant to our work, including Boolean Satisfiability, Constraint Programming, and Lazy Clause Generation.

2.1 Introduction

Classical Planning provides a formal framework for representing and reasoning about complex transition systems. It uses problem description languages inspired by propositional and first-order languages to describe states, actions, goals, and the preconditions and effects of each action. These languages allow for a clear and precise specification of the problem and the solution, as well as the possibility of verifying and analyzing the properties of the plan. Finding a plan is usually very hard in the worst case, and therefore, a lot of the research in this area is about how to make it practical.

Even in the classical sense, with deterministic and instantaneous actions and complete observability assumptions, many real-world problems can benefit from advancements in Classical Planning, including Logistics, Robotics, and Games. Solving Logistics, a complex problem, the intersection of assignment, routing, and sequencing problems, that comes from operations research, using classical planning allows us to control routes of packages and vehicles, map packages to modes of transportation, and order packages for transport. In robotics, we can use classical planning for motion and task planning, helping a robot plan how to move from one location to another while avoiding obstacles or picking up and delivering objects using its gripper. Moreover, while games generally require more expressive models representing partial observability and uncertainty, many can still be encoded into classical planning, especially Puzzle games, including Sokoban, Letter Boxed, and Rubik's Cube.

Not only does Classical Planning provide a rigorous and general framework for representing and solving deterministic planning problems, but it is also a basis for developing more realistic and expressive planning models. It has been extended and modified to accommodate more challenging aspects of real-world domains, such as uncertainty, durative actions, concurrency, resources, preferences, and multiple agents. These extensions build upon the basic concepts and methods of classical planning and often require novel solutions and approaches.

A key concept for Planning, similar to other areas in Artificial Intelligence, is the notion of Knowledge Representation and Reasoning. Knowledge Representation involves specifying a precise description, a model, of the world and the query about the world we have. Generally, we use formal languages to express the planning problems, where we expect the languages' semantics to be well-defined. On the other hand, methods to

planning reason over the mathematical description of the problem and attempt to answer the questions of plan existence and optimality.

Chapter Outline. First, we explain the classical planning model and how classical planning problems can be expressed using planning languages. The computational complexity of planning is then briefly discussed. After that, we explain the different computational approaches of planning, such as Planning as a search and Planning as a constraint satisfaction problem and the relevant algorithms for each. We end with a brief history of classical planning, as seen in international planning competitions.

2.2 The Model of Classical Planning

We can understand the classical planning model as a transition system, a directed graph whose vertices denote the set of states and whose edges are actions that force the state transition from the state represented by the source vertex to the destination state. Classical Planning is concerned with finding a path between the initial and one of the goal states in this graph. In other words, a plan, a sequence of actions in classical planning, is analogous to a path in the graph.

We now present a formal description of the classical planning model, which will help us explain the semantic interpretation of the planning languages.

Definition 2.1. The **transition system for a classical planning** problem P is defined as a tuple, $P := \langle S, s_0, S_G, A, T \rangle$, in which

- S is the state space, a finite and discrete set of states,
- $s_0 \in S$ is the initial state,
- $S_G \subseteq S$ is the set of goal states,
- A is the set of actions,
- $T : S \times A \to S$ is the transition function.

We define the set of *applicable* actions $A(s)$ as the subset of $a \in A$, for which $T(s, a)$ is defined, and the function $\mathcal{T} : S \times \Pi$, where Π is the set of action sequences, describes

the state resulting from the application of a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$ which we define recursively as

$$\mathcal{T}(s, \langle\rangle) = s \tag{2.1}$$

$$\mathcal{T}(s, \langle a_1, \ldots, a_i \rangle) = \begin{cases} T(\mathcal{T}(s, \langle a_1, \ldots, a_{i-1} \rangle), a_i), & \text{if } i > 1 \\ T(s, a_1), & \text{otherwise} \end{cases} \tag{2.2}$$

The above formulation requires that $\mathcal{T}(s, \langle a_1, \ldots, a_i \rangle)$ is defined for all $i \in \{1, \ldots, n\}$.

A solution to P is a sequence $\pi = \langle a_1, \ldots, a_n \rangle$ mapping the initial state s_0 to a goal state $s_n \in S_G$, $\mathcal{T}(s_0, \pi) \in S_G$, such that each action is applicable in the corresponding state, $a_i \in A(s_{i-1})$, along the induced sequence s_0,...,s_i,...,s_n where $s_i = T(s_{i-1}, a_i)$.

For Optimizing Planning, we would additionally have a cost function, $c : A \mapsto \mathbb{R}_0^+$, in the model that maps each action $a \in A$ to a non-negative cost. This allows us to define a total cost function as follows.

$$cost(\pi) = \sum_{i=1}^{n} c(a_i) \tag{2.3}$$

A plan π^* is in the set of *optimal* plans iff $cost(\pi^*) = \min_{\pi \in \Pi, \mathcal{T}(s_0, \pi) \in S_G} cost(\pi)$, i.e, the plan π^* has the minimal cost of all feasible plans.

2.3 Problem Description: The Languages of Classical Planning

Over the years, researchers in the planning community have developed various languages to describe classical planning instances. Most of these works are connected to the Stanford Research Institute Problem Solver (STRIPS) language [106], one of the most influential works in classical planning languages, motivated by the need for greater expressivity and properties that help the underlying computational approaches. In this section, we summarize the planning languages that the community uses and how they are mapped to the classical planning model. We describe STRIPS and PDDL, the standard planning languages in the International Planning Competition benchmarks, followed by a description of SAS$^+$, an important subset of Finite Domain Representation (FDR) tasks accepted by Fast Downward planner — a suite of state-of-the-art planning algorithms.

2.3.1 Stanford Research Institute Problem Solver (STRIPS)

Richard Fikes and Nils Nilsson developed the STRIPS language in 1971 at Stanford Research Institute [106] for representing and solving planning problems in artificial intelligence that involve finding a sequence of actions mapping an initial state of the world into a desired goal state. The name STRIPS stands for Stanford Research Institute Problem Solver, which is also the name of the automated planner they developed for solving planning problems. STRIPS was considered a novel language for modeling planning problems at the time. However, some thought it to be a specialization of *situation-calculus* with lower computational complexity [99]. Situation calculus is shown to be reducible to *theorem-proving* in first-order logic [62], which is generally *undecidable*. This original formulation of STRIPS lacked a precise formal semantics [45], which was subsequently refined and improved [90, 20].

The primary purpose of STRIPS is to describe the dynamics of a static world, which only changes when an operation is performed, as a transition system which is a directed graph, $\langle S, E \rangle$, where S is the set of world states comprising the nodes and E is the set of edges between them. In STRIPS, we describe the models of the *world* (states) as a collection of well-formed formulas that hold in a state, and a set of operators (actions) describe the allowed transitions, mapping between the states. An action a is a tuple, $\langle \text{Pre}_a, \text{Add}_a, \text{Del}_a \rangle$, namely, *precondition*, *additions list*, *deletions list*, which describe the constraints over the edges in the transition system. The precondition of an action is a well-formed formula that must hold in the source state, the deletions list is a set of well-formed formulas in the source model that may *not* hold in the target state, and the additions list those that must hold in the destination node.

For example, consider the classical problem of *Blocks*, illustrated in Figure. 2.1, where the *worlds* of Blocks consist of different arrangements of blocks that are possible within the physical limitations of the real-world and a robot is allowed the change the *world* state by executing actions. We can model the dynamics of Blocks world using the action schemas presented in Figure. 2.4, that model a *robot's* ability to *move* a block into a *new position* relative to an another *block* or *table*, when initially both *blocks* are *on* the *table*. We then ask whether a sequence of actions connecting two vertices exists. We separately capture the initial state as a collection of formulas (facts) that hold in the

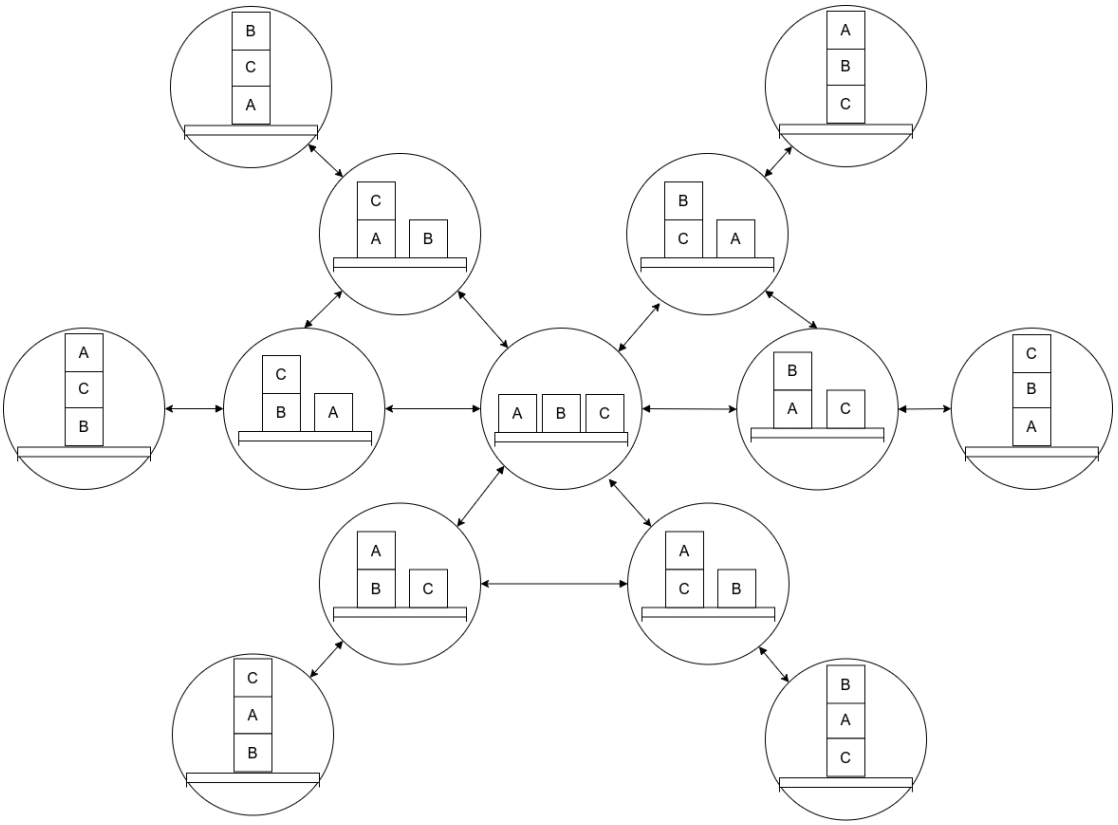


FIGURE 2.1: *Blocks* Transition System.

```
move(k,m,n):
    precondition = { clear(k), on(k, n), clear(m), k ≠ m ≠ n }
    delete−list  = { on(k,n), clear(m) }
    add−list     = { on(k,m), clear(n) }

moveToTable(k,m):
    precondition = { on(k, m), clear(k), k ≠ m }
    delete−list  = { on(k, m) }
    add−list     = { clear(k), on−table(k), clear(m) }

moveFromTable(k,m):
    precondition = { clear(k), on−table(k), clear(m), k ≠ m }
    delete−list  = { on−table(k), clear(m) }
    add−list     = { on(k,m) }
```

FIGURE 2.2: STRIPS operators, factored representation of the *Blocks* transitions.

initial state and the facts that must hold in the goal state, along with the set of all facts, as exemplified in Figure. 2.3.

We must note, however, that the physical constraints on the *Blocks'* worlds are only *partially* imbued in STRIPS description of Blocks. A curious observer would notice that the

```
initial-state: { on-table(A), on-table(B) }

goal-state:    { on(A, B) }

facts:         { on-table(A), on-table(B), on(A, B), on(B, A),
                 clear(A), clear(B) }

operators:     { move(A, B), move(B, A),
                 moveToTable(A, B), moveToTable(B, A),
                 moveFromTable(A, B), moveFromTable(B, A) }
```

FIGURE 2.3: STRIPS representation of the *Blocks* problem.

action schemata allow transitions from a state $s \coloneqq$ {on-table(A), on(A, B), on-table(B), clear(A), clear(B)} to $s' \coloneqq$ {on-table(A), on(A, B), on(B, A), clear(B)} when moveFromTable(B, A) is applied, both of which are inconsistent with the physical model of the *Blocks*. We could avoid the issue by selecting an initial state consistent with the world's physical model, after which the same action schemas ensure that only consistent states have a directed path from the initial state. A more straightforward approach would be to specify preconditions in a manner that disallows transitions from invalid world states of the *Blocks*, e.g., on-table(k) $\rightarrow \nexists$y, on(k,y). Remediation on a similar line of thought seems to be adopted by the community later by introducing the *timeless* predicates in PDDL [60], but the notion seems to have been discarded likely because the information is redundant in the mainline computational approaches based on the forward search from the initial state. This highlights the challenge of *efficiently* modeling the dynamics of the world in STRIPS, which is not straightforward for many real-world problems.

Next, we present the definition of the STRIPS problem and explain the precise semantics of the language with a mapping into the classical planning model.

Definition 2.2. We describe a classical planning problem compactly using **STRIPS** as the tuple $P \coloneqq \langle F, O, I, G \rangle$

- F is the collection of Boolean atoms or fluents
- O is the set of operators
- $I \subseteq F$ is a set of atoms that fully describe the initial state
- $G \subseteq F$ is a set of atoms in the goal state

The sets of atoms I and G have different meanings regarding the states in S. The set I represents only one state, in which the atoms in I hold, and the others are false by default — following the notion of "negation by failure". On the other hand, the set G represents a set of states in which the atoms in G hold, irrespective of the truth values of the other atoms.

We obtain a classical planning transition system $S(P) = \langle S, s_0, S_G, A, T \rangle$ from the STRIPS problem P as $S = 2^{|F|}$, $s_0 = I$, $s_G = \{s \,|\, G \subseteq s\}$, and T and A follow from the effects and preconditions of $a \in O$. Fikes and Nilson described the operator preconditions as a well-formed formula. However, we use the simplified formulation of propositional STRIPS presented by Bylander [20], which represents the precondition of actions as a set, i.e., a STRIPS action a is a tuple, $a := \langle \mathrm{Pre}_a, \mathrm{Add}_a, \mathrm{Del}_a \rangle$, where Pre_a is a set of atoms that must hold in the source state, Add_a the list of atoms that must hold in the target state, and Del_a the list of atoms that may not hold in target state. This allows for a precise definition of the Transition function using set operators, Eq. 2.4.

$$T(s,a) = \begin{cases} (s \backslash \{\mathrm{Del}_a\}) \cup \mathrm{Add}_a, & \mathrm{Pre}_a \subseteq s \\ s, & \textit{otherwise} \end{cases} \tag{2.4a}$$

$$A(s) = \{a \mid a \in O, \mathrm{Pre}_a \subseteq s\} \tag{2.4b}$$

An essential benefit of Propositional STRIPS is that it solves the problem of having to specify a large number of frame axioms, which exists in Situation Calculus, by representing a state as a set of facts instead of a logical conjunction of atoms and using set operators to define actions in STRIPS that also encode the frame axioms implicitly. However, we note that the state-based encodings of STRIPS into propositional logic, where the state is represented as propositional formulas, require specifying the frame axioms. This is avoided in causal encodings of STRIPS into Logic as the states are not defined explicitly, and therefore, the frame axioms are unnecessary.

2.3.2 Planning Domain Definition Language (PDDL)

The PDDL language [60] was invented to cater to the community's needs that wanted a *standard* but more *powerful* and *concise* modeling language for planning competitions. The language of STRIPS, although quite remarkable for its time, is limited in its

expressive power — not only determinism but also the assumption of *sequential* (non-concurrent) execution of *instaneous* actions, which Fikes and Nilsson found inadequate, in retrospective [45]. Furthermore, modeling constructs commonly available in other formal languages, like numeric variables and algebraic functions, are not available in STRIPS, requiring extensive effort to model specific problems. These challenges led to a multitude of developments in the field of planning languages, extending the language of STRIPS, including Pednault's Action Description Language [111], adding the ability to express complex transitions *concisely* by allowing new syntactic elements in action-schema definitions, e.g., conditional effects and universal quantification, and other changes to increase the expressive power of the language, e.g., Temporal-numeric Planning. Due to these developments, PDDL was born to allow researchers to define and add new building blocks to the language, facilitating fluid extensions. PDDL is a framework containing a *collection of languages*, each specified by the combinations of the *requirements* tags. This makes PDDL very flexible. However, that comes at the cost of standardizing the semantics of PDDL.

While there are issues with the semantics of some PDDL *requirements* [101], the community understands the commonly used extensions. One such extension is PDDL 1.2, which is widely used by the community to specify classical planning instances, including the benchmark instances of the IPCs we use for our experiments.

PDDL 1.2 While STRIPS and PDDL 1.2 represent the same class of problems — querying plans with specified characteristics over deterministic transition systems, syntactic elements of PDDL allow for a more concise and powerful representation. We present the semantics of PDDL 1.2 with an example from Blocksworld in, Figure 2.4 and Figure 2.5.

Blocks PDDL. We start by explaining the *domain* description in Figure 2.4. Note that the PDDL description is generally split into two components: the *domain* description, which specifies the set of possible facts and action schemas, and the *instance* description, which describes the initial state and the goal. The *domain* description includes the *:requirements* tags that capture the PDDL extensions. For example, the *:typing* tag enables many-sorted logic: we no longer assume that the universe is a collection of homogenous objects. We only need to define objects directly used in the action schema descriptions as *constants*. Then comes the *predicates*, which describe the different relationships that may exist between objects. We use them to capture *invariant* relations, and *fluent* predicates that capture the relationships that may change, like placing a block on the table

```
(define (domain blocks-3ops)
  (:requirements :equality :typing :negative-preconditions)
  (:types block - object)
  (:constants nothing - object)
  (:predicates
    (on ?b1 - block ?b2 - block)
    (on_table ?b1 - block)
    (clear ?b1 - block)
  )

  (:action move
    :parameters (?b_m ?b_from ?b_to - block)
    :precondition (and
      (clear ?b_m) (on ?b_from ?b_m) (clear ?b_to)
      (not (= ?b_m ?b_to)) (not (= ?b_m ?b_from)))
    :effect (and
      (not (clear ?b_to)) (not (on ?b_from ?b_m))
      (clear ?b_from) (on ?b_to ?b_m))
  )
  (:action moveToTable
    :parameters (?b_m ?b_from - block)
    :precondition (and
      (clear ?b_m) (on ?b_from ?b_m) (not (on_table ?b_m))
      (not (= ?b_m ?b_from)))
    :effect (and
      (not (on ?b_from ?b_m)) (on_table ?b_m) (clear ?b_from)
    )
  )
  (:action moveFromTable
    :parameters (?b_m ?b_to - block)
    :precondition (and
      (clear ?b_m) (clear ?b_to) (on_table ?b_m)
      (not (= ?b_m ?b_to)))
    :effect (and
      (not (on_table ?b_m)) (not (clear ?b_to))
      (on ?b_to ?b_m))
  )
)
```

FIGURE 2.4: Blocks domain description in PDDL.

or on another block. Finally, we describe the action schemas, which define the possible transitions. In each action description, the *precondition* is a logical formula, and *effects* describe the set of atoms that must hold in the state obtained through the application of the action. Note that the keyword "not" specifies a negative atom. The *instance* PDDL, Figure 2.5, captures instance-specific data, including the set of objects (blocks),

the initial state of those blocks, and the expected goal state.

```
(define (problem blocks-easy)
  (:domain blocks-3ops)
  (:objects A B - block)
  (:init (clear A) (clear B) (ontable B)
    (ontable A) (handempty))
  (:goal (and (on A B)))
)
```

FIGURE 2.5: Blocks instance description in PDDL.

The other common extensions include **PDDL 2.1**, which Fox and Long [48] developed for the 3^{rd} international planning competition, offering support for numeric fluents and durative actions. **PDDL 3.0** and **PDDL 3.1** add support for preferences, trajectory constraints, and function symbols. Another extension, **PDDL+**, supports process and continuous changes. In detail, we will look into PDDL 2.1 semantics in Chapter 3. We will not discuss other extensions as they do not concern with the problems we address in our work.

2.3.3 SAS$^+$

SAS$^+$ [4] is an extension of Simplified Action Structure (SAS), which is like propositional STRIPS except that it allows for multi-valued variables. While the planning benchmarks are generally encoded in STRIPS, the SAS$^+$ representation has its benefits. The multi-valued representation implicitly captures exactly one condition over a subset of atoms, which in certain tasks allows for a more concise representation of a state that is an order of magnitude smaller. Furthermore, it enables the computation of tighter heuristics when the domain of variables is non-binary [70]. For this reason, FAST DOWNWARD [67] translates PDDL into a SAS$^+$ representation, allowing it to tap into the benefits of the SAS$^+$ formalism.

2.3.4 Complexity of Classical Planning.

Given a classical planning problem, we typically ask two types of questions: one is that of PLANEXISTS(P) and the other is that of PLANCOST(P, k), associated with *satisficing* and *optimizing* planning, respectively. **PLANEXISTS(P)** asks: *Does a plan π for P exists?*. On the hand, **PLANCOST(P, k)** is concerned with the question: *Is there a plan*

π with cost less than or equal to k that satisfies P?. We note that the complexity class of the problem depends on the modeling language. Hence, the computational complexity varies based on the exact model used to represent the classical planning problems. The complexity class of these decision problems is PSPACE-complete in the standard STRIPS representation [20]. Bylander also presents special cases of STRIPS seen as easily identifiable modifications to the grammar of the language of lower complexity: NP-complete and polynomial. Another similar work has shown complexity results for the SAS$^+$ planning formalism [4]. Erol et al. [41] presented complexity results for variation in STRIPS specifications, asserting that the complexity depends on the nature of the planning operators and ranges from constant time to EXPSPACE-complete. All these results point to the fact that Classical Planning, in the general case, is intractable.

2.4 Computational Approaches

In this section, we cover two prominent computational approaches: Planning as Search and Planning as Constraint Satisfaction, to solve planning problems and some major previous works in this direction related to our research.

2.4.1 Planning as Search

In *Planning as Search* approach, we think of the model of Planning as a transition system, a graph with states as vertices and transitions as edges, and hence, planning is reduced to state-space search, in which an algorithm systematically explores a graph of possible states and transitions to find a path from the initial state to one of the goal states. Any graph traversal algorithms could be used for this purpose, including *breadth-first search* (BrFS), *depth-first search* (DFS), and even cost-optimal paths could be found using *Dijkstra's* algorithm [33]. All these algorithms have *low polynomial* runtime complexity in the graph's size; hence, it would seem that we already have performant approaches. But in planning, we use *factored representations* like STRIPS to specify the transition system where the size of the transition system is *exponential* in the number of fluents in the representation. This poses a fundamental challenge as the complexity of these algorithms is not polynomial but exponential in the size of the planning problem. Therefore, these algorithms do not automatically scale well on planning instances.

An approach to handle the challenge is using *informed search* methods in which the algorithm uses information about the goal to *guide* the search. The notion of *informed search* stands in contrast with algorithms such as BrFS, DFS, and Uniform Cost Search (Dijkstra's), which are considered *blind*, as they do not use additional information. Many performant *informed search* algorithms have been designed over the years, most of which use a *heuristic function* [110, 38] to guide the search, including A* [65], IDA* [86], MA* [146] which are *complete*, and also, *incomplete* algorithms like hill-climbing [132]. In the next section, we delve deeper into the heuristic search approach to planning, which is also our research's primary subject of investigation.

2.4.1.1 Heuristic Search

Incremental heuristic search over transition systems has formed the backbone of many state-of-the-art planning systems over the years [15, 73, 67, 120, 145]. As the name suggests, these algorithms use a heuristic measure that estimates the remaining cost to the goal to guide the search. Of the many frameworks of heuristics search, *best-first search* is the most prominent.

Best-First Search is an informed search algorithm that explores a graph by expanding the most promising node according to a specified ranking function starting from the initial state s_0. A node in Best-First Search (BFS) carries necessary information about how it was reached. It is typically defined as a tuple $(s, g(n), n_{\text{parent}})$ capturing the state s, the cost of the path to the node $g(n)$, and parent node n_{parent}. A ranking function f is a linear combination of the accumulation cost to the node $g(n)$ and the estimated cost to the goal $h(n)$, i.e., $f(n) = v \cdot g(n) + w \cdot h(n)$. A smaller $f(n)$ value is typically considered better, and the node with the smallest $f(n)$ value is expended next. BFS maintains an *open* and *closed list* to track the candidate nodes for expansion and already expanded nodes, respectively. The search executes iteratively, selecting the best candidate from the *open list* for expansion and generating its successors by applying all applicable actions, incrementally developing a spanning tree of the transition system rooted in s_0.

Standard search algorithms such as A*, Greedy Best-First Search, and Uniform Cost Search are all covered within this framework. They can be obtained by simply substituting the parameters v and w with suitable constants.

For the **A* algorithm**, we set $v = 1, w = 1$ as we want to expand the node with the minimum value of the evaluation function $f(n) = g(n) + h(n)$, breaking ties with the $h(n)$ value, smaller is better [65]. A* is popular in *optimizing* planning as it returns *optimal* plans when provided with a non-negative cost function and a heuristic function that is *admissible*. If the heuristic is *consistent*, the evaluation function f monotonically increases along the nodes on a path. The condition of admissibility requires that $\forall n, h(n) \leqslant h^*(n)$, where h^* is the optimal cost to the goal from node n. Consistency holds iff for all transitions (n, n') in the transition system, $h(n) \leqslant cost(n, n') + h(n')$, where $cost(n, n') = c(a)$, and a is the action responsible for the transition (n, n'). Moreover, an extension of A* with *duplicate detection* and *reopening* produces optimal plans with only *admissible* heuristics. A node n is considered a duplicate when the state in the node is already captured by a node in the closed list. Without a guarantee of f being monotonic, a shorter path to state in n is possible. Hence, *reopening* the node in the closed list and comparing its cost against the path leading to n is necessary to give an optimality guarantee. In this case, the node n is expanded iff it has a smaller cost, in contrast to when the heuristic is consistent, where the duplicates can be discarded immediately.

A common variation of A* is the **WA* algorithm** where the weight $w > 1$ [115] are allowed, making the method *sub-optimal*, although the cost of suboptimal plans is bounded by the factor w. The algorithm is popular in the community when an optimality guarantee is unnecessary or near-optimal results are sought, as it empirically results in faster solutions as the search more greedily targets the goal. An approach to start with a high value of w and slowly reduce it to 1 [120] is commonly used for *anytime* planning, including in our submission to the IPC 2023 *satisficing* track.

Similarly, **Greedy Best-First Search** and **Uniform Cost Search** can be obtained by setting $v = 0, w = 1$ and $v = 1, w = 0$, respectively. These parameter choices result in two search algorithms with contrasting properties, as Greedy Best-First Search (GBFS) is wholly focused on getting closer to the goal at any time in the search without considering the cost of the path. In contrast, Uniform Cost Search (UCS) only minimizes the accumulated cost. Hence, GBFS is suited for *satisficing* planning and is generally quite performant. UCS is optimal.

2.4.1.2 Domain Independent Heuristics

Heuristic functions are a crucial factor impacting the performance of the Best-First Search algorithms. While it is probably preferable to design *domain-specific* heuristics when dealing with a particular class of problems, this approach does not work for the general set of benchmarks the planning community is interested in, including the planning competitions. Hence, the research in planning as heuristics search is mainly concerned with general *domain-independent* heuristics, which, as the name suggests, do not depend on the domain of the planning instance. That does not mean that these heuristics aren't tailored to specific structures of planning instances, just that no assumptions are made on the existence of predicates, actions, and objects from a particular domain.

There are two main types of domain-independent heuristics. *Analytically derived heuristics*, including *delete-relaxed*, $h_{\max}$ [15], h_{add} [73], h_m [58] and LM-Cut [69], and *learning based heuristics* like STRIPS-HGN [138]. Analytically derived heuristics typically use some form of relaxation or abstraction of the original planning problem, such as ignoring the deletions list of actions and grouping similar states. On the other hand, learning-based heuristics are obtained by training a machine learning model on a training set of solved planning problems using features extracted from the states and the action description. Both form separate fields within the planning community; we are only concerned with analytical heuristics. Next, we explain the standard *domain-independent* heuristics that form the baseline in planning competitions.

Perfect Delete Relaxation heuristic. This heuristic is the most common one and also easy to understand. We simply forget the deletions list in the STRIPS operators, resulting in a *delete-relaxed* problem where a solution can be found in *polynomial* time. However, obtaining an *optimal* plan remains NP-complete [20]. The optimal cost from a state in the *delete-relaxed* problem h^+ comprises the perfect delete relaxation heuristic.

Definition 2.3. A **Deleted-relaxed** problem of a STRIPS problem, $P := \langle F, O, I, G \rangle$, is $P^+ := \langle F, O^+, I, G \rangle$, where the operators O^+ are derived as

$$O^+ = \{\text{Pre}_o, \text{Add}_o, \varnothing \mid o \in O\}$$

The delete-relaxed heuristic is obtained as $h^+(n) := cost(\pi^*_+)$, where π^*_+ is an optimal-cost solution to P^+ that uses the state captured by node n as the initial state.

$h_{\max}$ heuristic. The h^+ heuristic is NP-complete, which implies that computing it is likely intractable. This is not ideal for heuristic search, as we want *fast heuristics*, much faster than the complexity of planning itself, ideally *low polynomial*. The $h_{\max}$ heuristic addresses this issue by estimating the accumulated cost to achieve a state using the maximal cost to achieve *any atom* of the state, relaxing the problem even further. The resulting heuristic function is defined as follows.

$$h_{\max}(s) \coloneqq \max_{q \in G} \; h_{\max}(q, s) \tag{2.5}$$

where the function $h_{\max} : 2^F \times 2^F \mapsto \mathbb{R}_+^0$ is defined recursively.

$$h_{\max}(p, s) \coloneqq \begin{cases} 0, & \text{if } p \subseteq s \\ \min\limits_{a \in O, p \in \mathrm{Add}_a} \; c(a) + h_{\max}(\mathrm{Pre}_a, s), & \text{else if } |p| = 1 \\ \max\limits_{q \in p} \; h_{\max}(\{q\}, s), & \textit{otherwise} \end{cases} \tag{2.6}$$

Notably, $h_{\max}$ is *admissible*, allowing for the construction of an *optimizing* method with A*. It is also computable in polynomial time using a dynamic programming approach.

$h_{\mathbf{add}}$ heuristics. While $h_{\max}$ assumes that the atoms achieved with equal costs are achieved together, h_{add} takes a contrary position that atoms are achieved in sequence. It is defined as follows.

$$h_{\mathrm{add}}(s) \coloneqq \sum_{q \in G} \; h_{\mathrm{add}}(q, s) \tag{2.7}$$

where the function $h_{\mathrm{add}} : 2^F \times 2^F \mapsto \mathbb{R}_+^0$ is defined recursively.

$$h_{\mathrm{add}}(p, s) \coloneqq \begin{cases} 0, & \text{if } p \subseteq s \\ \min\limits_{a \in O, p \in \mathrm{Add}_a} \; c(a) + h_{\mathrm{add}}(\mathrm{Pre}_a, s), & \text{else if } |p| = 1 \\ \sum\limits_{q \in p} \; h_{\mathrm{add}}(\{q\}, s), & \textit{otherwise} \end{cases} \tag{2.8}$$

The h_{add} heuristic is not guaranteed to be admissible. Therefore, we cannot use it for optimizing, but it is well suited for satisficing planning.

h^m family of heuristics. The h_m heuristics generalize the $h_{\max}$ heuristic, considering the cost of achieving a set of atoms of cardinality less than or equal to m together. When $m = 1$, h^1 is the same as $h_{\max}$. When $m = 2$, the highest cost of achieving any atom pair

is used, and so on. Computing h^2 for each node can be impractical in large problems. However, it is still useful as a preprocessing tool to compute *mutex pairs* — atoms that cannot occur together in any state, i.e., pair $\langle p, q \rangle$ is mutex when $h^2(\langle p, q \rangle) = \infty$ [2]. The h^m heuristic is defined as follows.

$$h^m(s) \coloneqq \max_{q \subseteq G} \; h^m(q, s) \tag{2.9}$$

where the function $h_{\text{add}} : 2^F \times 2^F \mapsto \mathbb{R}^0_+$ is a recursively defined.

$$h^m(p, s) \coloneqq \begin{cases} 0, & \text{if } p \subseteq s \\ \min\limits_{a \in O, \text{Add}_a \cap p \neq \emptyset, \text{Del}_a \cap p = \emptyset} \; c(a) + h^m(p \backslash \text{Add}_a \cup \text{Pre}_a, s), & \text{else if } |p| \leqslant m \\ \max\limits_{q \subseteq p} \; h^m(q, s), & \textit{otherwise} \end{cases} \tag{2.10}$$

LM-cut heuristic. A *landmark* is a condition that must hold in all plans. The literature discusses two types of landmarks: *Fluent* and *Action* landmarks, which are formulae over a set of facts and actions, respectively. The LM-cut heuristic, $h^{\text{LM-cut}}$ [69], is an *admissible* heuristic that uses the notion of *action landmark*. It is commonly used for optimal classical Planning with BFS, and we use it as a baseline to compare our work on Lifted Sequential Planning. $h^{\text{LM-cut}}$ computation uses $h_{\max}$ with $h^{\text{LM-cut}} = h_{\max}(s)$ if $h_{\max}(s)$ is 0 or ∞. Otherwise, the cost is non-zero, and the heuristic is computed using a disjunctive action landmark of the delete-relaxed problem. The action landmark constitutes a set of actions with non-zero cost, such that every relaxed plan includes at least one action. The exact computations steps for $h^{\text{LM-cut}}$ are non-trivial, and since it is not a direct subject of our research, we request the reader to refer to work by Helmert [69] for details.

2.4.1.3 Width-based Search

In this section, we explain, in detail, the family of width-based search algorithms, which forms an essential subject of our research on tractable approximations discussed in Chapter 4. We begin by explaining the measures of *problem width* and *state novelty* that are central to these algorithms.

Definition 2.4. The **width** of a STRIPS planning instance, $P \coloneqq \langle F, O, I, G \rangle$, is defined as the minimum value m such that the tuple graph G^m contains a tuple of atoms $t \supseteq G$, where $G^m = \langle V^m, E \rangle$, V^m is the set of tuples of atoms of size no greater than m, and there exists an edge (t, t') iff for each optimal plan π for $P(t) \coloneqq \langle F, O, I, t \rangle$, there exists an action $a \in O$, s.t. the plan obtained by concatenating π and a is optimal for $P(t')$.

Lipovetzky and Geffner [91] proved that a planning instance with the width measure m can be optimally solved in time exponential on m. Next, we explain an algorithm that exploits these results when m is small.

Iterated Width (IW) The IW algorithm involves iterative calls to IW(k), $k = (1, \ldots, |F|)$ until the problem is solved, where a call to IW(k) is guaranteed to produce an optimal solution if the width of the problem is no greater than k. The time complexity of IW(k) is $O(|F|^k)$. IW(k) is designed as *breadth-first search* with a minor difference: the successor states that do not pass a *novelty test* are pruned. The test uses the notion of *state novelty*.

Definition 2.5. The **novelty** value of state s is the size of the smallest *new* tuple in the set of all possible tuples (conjunctions) of atoms in s. The value is $|F| + 1$ if no such tuple exists. A tuple t is considered *new* if none of the states generated before s include the tuple t, i.e., $\forall s' \in \mathcal{P}(s), t \nsubseteq s'$, where we denote the set of states generated before s as $\mathcal{P}(s) = \{s' \,|\, e(s') < e(s)\}$, and the mapping $e : S \mapsto \mathbb{N}$ ranks a state in the order of expansion.

Theorem 2.6. *The time and space* complexity *of novelty computation for a state s with novelty value k is $O(|F|^k)$.*

Proof. To test whether a tuple t of size k is the smallest *new* tuple in the set of all possible tuples of atoms in s, one must enumerate all the tuples of size less than k and verify it against a database storing the previously seen tuples. The count of tuples of size no greater than k is $\sum_{i=1}^{k} \binom{|F|}{i}$. Hence, the count of checks involved and the database size is $O(|F|^k)$. □

In IW(k), the *novelty test* requires that the novelty value of a state s is no greater than k. The node is pruned if the test fails, i.e., we treat the states with novelty value greater than k as duplicates. As an example, if $t = (p, q)$ is the only new tuple in s, the novelty

value is 2, but if there is another new tuple $t' = (p)$ in s then the novelty value is 1. It is important to note that in the absence of prior knowledge of the width of a problem, IW is designed to target problems with a small width. Therefore, perhaps a more important question is what kind of problems have low width. Lipovetzky and Geffner [91] showed that when the goal set G is restricted to a single atom, most of the benchmark instances of the IPCs appear to have low width, empirically. However, since the goals in most planning problems require many atoms to hold in the final situation, IW generally does not automatically scale on planning instances in their default specification. Much effort has been put into research to handle this challenge, including Serialization of the planning problem [94], and adopting Best-First Search to do *exploration* using novelty to guide the search [92, 93].

Serialized Iterated Width. A prominent work addressing the challenge IW faces is the Serialization of a classical planning problem, which assumes that the goal atoms could be achieved in a particular sequence, allowing for decomposing the problem P into a sequence of subproblems. Of course, such an ordering is unknown a priori, so the question here is how to order the goal atoms. Serialized Iterated Width (SIW) avoids explicitly answering this question by using IW to both serialize the problem $P := \langle F, O, I, G\rangle$ and find a path in each of the subproblems. It solves a sequence of problems $P_k := \langle F, O, I_k, G_k\rangle$, where $k = 1, \ldots, |G|$, $I_1 = I$, $I_k, k > 1$ is the state in which G_{k-1} is achieved, G_k is the *first* set of atoms achieved from I_k using IW such that $G_{k-1} \subset G_k \subseteq G$ and $|G_k| = k$. Hence, the problem is decomposed into a sequence of planning problems where goal atoms are sequenced in the order IW finds them. The k^{th} iteration of IW finds a path to a goal that contains atoms of G_{k-1} and one additional atom from G. As we can see, SIW does not rely on any heuristic guidance, and it solely depends on the assumption of the width of the problem structure. Yet, the approach has shown performance comparable to established heuristic methods [91].

Best-First Width Search. Since the notion of width and novelty is orthogonal to that of standard heuristics used in planning, it is to no surprise that many works seek to integrate the two [93, 92, 30]. Best-First Width Search (BFWS) is an effort in that direction. BFWS is a family of algorithms where the evaluation function of BFS takes a slightly different form and has shown state-of-the-art performance in the *satisficing* and *agile* track of the IPC [116]. We defer the detailed discussion on BFWS to Chapter 4, where we showcase our work on Approximate BFWS.

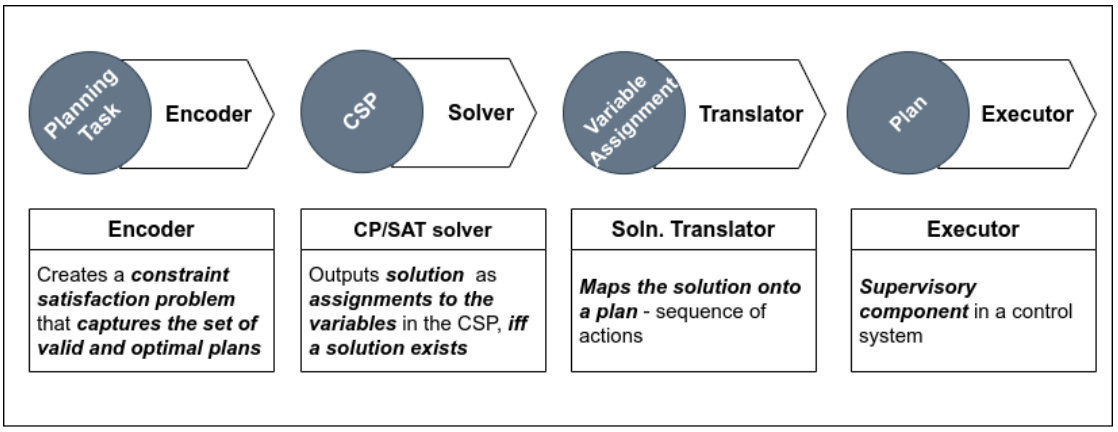


FIGURE 2.6: *Workflow* of Planning as Constraint Satisfaction.

2.4.2 Planning as Constraint Satisfaction

One of the most significant advances in classical planning was the realisation of Green's vision for theorem proving [62] as a framework for general problem-solving via the groundbreaking seminal work of Kautz & Selman [84, 83]. Putting together the insights of Blum and Furst [12], where Planning becomes the problem of analyzing whether goal states are reachable in a suitably defined graph, with space efficient solutions to the *frame problem* formulated in the Situation Calculus [100, 63], Kautz & Selman showed that formulating planning problems in terms of the satisfiability of Conjunctive Normal Form (CNF) formulas was feasible and, at the time, highly scalable. Attention to Planning as satisfiability has been somewhat eclipsed since then with the development of planning algorithms based on direct but lazy, incremental heuristic search over transition systems [15, 73, 67, 120, 145]. Yet deep theoretical connections exist between Planning as satisfiability and as heuristic search [56, 123] questioning [125, 143] perceptions of either approach as being parallel or mutually exclusive. In this section, we explain the notion of planning as constraint satisfaction, covering both *Constraint Satisfaction Problem (CSPs)* and its boolean specialization — *Boolean satisfiability (SAT)*, and important previous works in this direction that have inspired our research on CP Encodings of Sequential Planning.

Planning as constraint satisfaction is a way of solving planning problems using techniques from the field of constraint satisfaction, which, if considered pedantically, only involves approaches of Constraint Programming but could also include the approaches in the fields of Boolean satisfiability, Mixed Integer Linear Programming, Satisfiability

Modulo Theory, and Answer Set Programming, each of which is a specific field of constraint satisfaction. Figure 2.6 illustrates the steps involved in planning as constraint satisfaction, requiring encoding Planning into CSP/SAT as the first step, which can then be solved using general-purpose solvers. However, there are many challenges in designing a planner within this framework. For one, unlike planning languages, the representation of Constraint Satisfaction Problems does not have a way to encode action schema constraints directly, therefore requiring the specification of a complex set of constraints on variables representing a solution sequence — a plan. Furthermore, since the plan length is unknown, the encoders tend to bootstrap the encoding with an *assumption* on the upper bound on the number of actions (edges in transition system) in the plan, and increasing the bound when necessary, commonly known as *iterated* CSP solving. Moreover, it seems that there are many possible encodings of planning into CSP with starkly different performances, which also vary with the choice of solving technology, usually treated as black boxes, making developing performant encodings rather tricky.

In this, Kautz & Selman [84], and then Kautz & Selman & McAllester (KSM) [83] were first to attempt to address these challenges, developing a practical reduction of Planning into SAT and presenting several encodings for planning problems with STRIPS description and examining the size complexity of the different encodings in terms of the number of variables and the resulting SAT formulas, which was quite performant for those times. They also proposed a new nonlinear *causal* encoding based on the theory of causal planning that uses the concept of "lifting" from the theorem-proving, allowing for further reductions in the number of variables used by the encoding by eliminating either state or action variables. The authors initially [83] suggested that the *causal* encoding strictly dominates the others in terms of asymptotic complexity, but later [135] concluded that in practice, the performance of the *causal* encoding appeared to be worse than the *state-based* encoding. We request the reader to refer to the original KSM paper [83] to understand the encoding in detail. We present an encoding of Lifted Planning into CP in Chapter 5 that is closely related to the KSM encoding.

Afterwards, the approach seems to have been eclipsed by planning as heuristics search [14] with the many efforts to encode planning as CSP producing mixed results [35, 104, 149] until Rintanen [128, 125, 123] put it back in the limelight with runner-up award in the *agile* track of 8^{th} International Planning Competition [126]. Rintanen accomplished this through a long and exhaustive research endeavor that spanned over a decade, which

involved developing efficient encodings of Planning based on $\exists-$step semantics [128], parallelized/interleaved search strategies [127, 128], planning specific heuristics [125], binary mutex computations [122] and a specialized solver for large planning instances [124]. The work demonstrates that Planning as satisfiability, despite appearing simple after an encoding is created, requires significant work to be made performant in typical planning instances found in the IPCs. A later effort to refine Rintanen's work, FreeLunch [116], did not achieve the same kind of success.

2.5 Constraint Satisfaction

Constraint satisfaction problems (CSPs) involve finding an answer that satisfies a set of constraints or relations, widely used to model combinatorial optimization problems arising in artificial intelligence and operations research. This section introduces the basic concepts of CSPs, discusses some examples and applications, and reviews prominent computational approaches for solving CSPs.

Definition 2.7. A **Constraint Satisfaction Problem** [131] is defined as a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, consisting of a finite set of variables $\mathcal{X} = \{x_1, \ldots, x_n\}$, where the domain of the variable x_i is defined given by i^{th} element of the tuple $\mathcal{D} = (D_1, \ldots, D_n)$, and $\mathcal{C} = (C_1, \ldots, C_m)$ is a tuple of constraints. Each constraint C_i is a pair (S_i, R_i) consisting of a scope and a relation, where the scope $S_i \subseteq \mathcal{X}$ defines the dependent variables, and $R_i \subseteq \times_{i \in [S_i]} D_i$[1] is a subset of the Cartesian product of the domains of variables in the scope.

A **solution** to a CSP is an assignment to variables x_i, $\bar{\sigma} = (\sigma_1, \ldots, \sigma_n)$, such that $\sigma_i \in D_i$, and each constraint C_i of C is satisfied, in that the projection of $\bar{\sigma}$ onto the scope S_i is contained in R_i, i.e.,$(\sigma_{[S_i]_1}, \ldots, \sigma_{[S_i]_p}) \in R_i$. The CSP is *unsatisfiable* if no such assignment exists.

2.5.1 Constraint Programming

Constraint programming (CP) is a paradigm for solving combinatorial problems framed as a *constraint satisfaction problem* (CSP) in which the user defines a task as a set of variables, domains, and constraints and lets a general-purpose search algorithm find a

[1] We use $[\mathcal{X}]$ to denote the *ordered indexing* of the elements of set $\mathcal{X}$, e.g., $[\mathcal{X}] = (1, \ldots, n)$ for $\mathcal{X} = \{x_1, \ldots, x_n\}$

solution that meets all the constraints. It has been successfully used to solve various problems in operations research, including scheduling, planning, resource allocation, and many other optimization problems [142, 131]. We can also view it as a form of declarative programming where the user specifies the problem in terms of what needs to be satisfied rather than how to satisfy it. It is strongly influenced by logic programming, rooted in Constraint logic programming [78] where the user defines the problem as logical relations, and the solver uses inference rules to derive a solution. This section briefly overviews the CP approach to solving CSPs, which we use extensively in Chapters 5, 6, and 7.

The CP approach to solving CSPs involves various techniques, such as constraint propagation, domain filtering, branching, backtracking, and heuristics to explore the search space and prune the infeasible or suboptimal regions. A solver can also use hybrid methods that combine CP with other paradigms, such as satisfiability, mathematical programming, local search, or metaheuristics, a few of which we will cover in detail in Chapter 6. Central to the research in CP is finding efficient methods for enforcing constraints via propagation methods, where the key is to design fast methods to enforce the *local consistency* conditions — for a subset of constraints, typically one constraint. For example, if two variables $x_1 \in \{1, \ldots, 10\}$ and $x_2 \in \{1, \ldots, 100\}$ are in the scope of constraint $x_1 + x_2 = 10$, then, we can prune D_1 and D_2 to $\{1, \ldots, 9\}$ since the remaining values are inconsistent. There are different types of constraint propagation, depending on the level of consistency that they enforce. One of the most basic types of consistency is node consistency, which ensures that each variable has at least one value in its domain that satisfies its unary constraints (constraints that involve only one variable). For example, given the constraint $x \leqslant 1$, and that the domain of x is $\{-2, -1, 1, 2\}$, we can infer that 1 and 2 are inconsistent with the constraint and remove them from the domain of x, assuring node consistency.

A stronger type of consistency is arc consistency, which ensures that for each pair of variables that share a binary constraint, every value in the domain of one variable has a compatible value in the domain of the other variable. Arc consistency is generalized into domain consistency condition or generalized arc consistency, which includes non-binary constraints. To further strengthen the consistency, we can use path consistency, which ensures local consistency for each triple of variables that share two binary constraints, i.e., every pair of values in the domains of two variables has a consistent value in the domain of the third variable. Other types of consistency, such as k-consistency and

global consistency, generalize the notion of consistency to larger subsets of variables and constraints. However, enforcing higher levels of consistency is usually complex, requires more computation time and memory, and may not be worth the effort if there is only a slight reduction in search space. Therefore, the choice of the appropriate type of consistency depends on the characteristics of the problem and the availability of efficient inference methods.

We also note that while Constraint propagation is a powerful and efficient inference method for solving CSPs, it is not always sufficient to find a solution or prove unsatisfiability. The constraint propagation would often reach a point where no more domain reduction can occur, but the solution is unknown. In these cases, we must combine constraint propagation with a search method, such as backtracking or branch and bound, that explores the remaining search space, trying different assignment values to the variables. Generally, the search method invokes constraint propagation at each step to prune the search space further and improve the performance. For a broader study of typical constraints in CP and inference methods used for constraint propagation and domain reduction, we would request the readers to refer to the works of Marriot et al., Hooker, and Rossi et al. [97, 74, 131].

Another framework for solving combinatorial problems closely related to the CP approach to solving CSPs is that of *Boolean satisfiability* (SAT). The problem of SAT is a particular case of constraint satisfaction, where the variables are *all* Boolean, and the constraints are clauses. However, each of these approaches evolved independently until recently, when the CP community began to compare, contrast, and integrate the two [153, 7, 108, 114]. In the next section, we explain the SAT paradigm, which forms the core of the Google's CpSat solver that we extensively study in our research.

2.5.2 Boolean satisfiability

Definition 2.8. A **Boolean satisfiability** problem determines whether there exists a truth assignment $\bar{\sigma}$ to the set of Boolean variables $\mathcal{X} = \{x_1, \ldots, x_n\}$ that satisfies the propositional formula ϕ, where each literal l_i in ϕ is associated with a Boolean variable x_i such that l_i holds *iff* $x_i = true$.

An assignment **satisfies** the formula ϕ, *iff* the result of substituting the literals l_i in ϕ with the truth assignment of variables x_i evaluates to *true* in the model of propositional logic.

Both CSP and SAT are NP-complete [28, 87], and both can solve many of the same classes of problems. However, the computation methods in both areas developed separately, allowing for *orthogonal* approaches to emerge over time, leaving a potential for integration. Indeed, even though the field of satisfiability continues to develop independently, researchers in the past two decades have not only identified SAT encodings of some CSP problems that outperform conventional CSP methods [114] but also integrated the approaches in *satisfiability* with those used for CSP solving, benefiting from the best of both worlds [113, 142]. One of the most prominent approaches towards this end is *Lazy Clause Generation* [108], which combines Boolean satisfiability with Finite Domain propagation.

Lazy Clause Generation (LCG). LCG exploits the advantages of both CP and SAT, tapping into the efficient inference methods and search procedures in SAT while allowing the users to express the problem as a more flexible CSP. The power of LCG lies behind its ability to represent a complex system of constraints *implicitly* — without requiring a large number of equivalent clauses in the SAT formula, significantly reducing the burden on the SAT engine due to reduced memory footprint, and also, on the modeler. We explain the concept of LCG with a simple example. Consider the constraint $x_1 + x_2 \leqslant 1\,000\,000$, where $x_1, x_2 \in \{0, \ldots, 1\,000\,000\}$. We could encode this constraint equivalently in SAT by creating a Boolean variable $b_{ij} \iff (x_i > j)$ for each value $j \in D_i$ and adding clauses $\neg b_1 j \vee \neg b_{2(1\,000\,000-j)}$. Even this trivial constraint that a random assignment to the variables could satisfy in 1 out of 4 attempts requires over a million clauses in the SAT formula. This is where LCG comes into the picture. It allows the constraint $x_1 + x_2 \leqslant 1\,000\,000$ to be represented implicitly using a finite domain propagator that generates clauses of the form $\neg b_1 j \vee \neg b_{2(1\,000\,000-j)}$ lazily, only to enforce local consistency. The approach has tremendous potential in problems where a small number of such clauses must be made explicit to find a solution. Indeed, when integrated into the framework of Conflict-driven clause learning (CDCL), the approach has shown state-of-the-art performance [142]. We seek to capitalize on this technology in our experiments, extending the implementation of LCG in CpSat solver from Google's OrTools package [113].

2.6 International Planning Competitions

The International Planning Competition (IPC) is an event that aims to evaluate and advance the state-of-the-art in automated planning systems. The IPC is organized in the context of the International Conference on Planning and Scheduling (ICAPS), a leading venue for researchers and practitioners in Planning and Scheduling, which has also helped to develop and spread planning technology and discover new challenges and opportunities for planning research. The IPC has various tracks that target different aspects or subfields of planning, including the Classical Track, one of the oldest and most conventional tracks. It started in 1998 and has drawn many participants from academia and industry.

As the name suggests, the Classical Track focuses on Classical Planning. Its goal is to promote the advancement and evaluation of planning methods, measuring the state-of-the-art approaches in planning and finding new challenging benchmarks. It is a significant event for the community and a valuable source of information and inspiration for anyone interested. Typically, the track has four sub-tracks: **Optimal Track**, in which the planning systems must find an optimal plan for each problem or report that no such plan exists, scored based on the number of problems solved and the time to solve them, **Bounded-Cost track** requires a plan whose cost is below a given threshold or report that no such plan exists, the systems are scored based on the number of problems solved, the quality of the plans found, and the time to solve them, **Satisficing track** in which the systems are scored based on the number of problems solved, the quality of the plans found, and the time to solve them, and lastly, **Agile track**, which focuses on finding a satisficing plan for each problem as quickly as possible. The diversity of tracks allows the community to assess the performance of algorithms in various settings of the environment(computational) and the objective. Next, we present a brief review of the state-of-the-art approaches in planning over the years through the lens of the IPCs.

IPC 1998, 2000, 2002, 2004. The International Planning Competition grew rapidly in the first few IPCs, from five contestants in 1998 to twenty in 2004. Heuristic search planners were the dominant approach for Classical Planning in these competitions, with HSP, FF, and FD planners [13, 15, 14, 73] performing very well. HSP uses the hill-climbing search strategy with restarts, using h_{add} heuristic for guidance and restarting whenever it gets stuck at local minima. On the other hand, FF, firstly, does an incomplete

hill-climbing search that only considers helpful actions in the relaxed plan generated along with h_{FF}. Then, uses a greedy best-first search driven by h_{FF}. FAST DOWNWARD (FD) [67], the winner of the fourth IPC, is also a heuristic search planner. It is a highly efficient implementation of SAS$^+$ formalism extended with axioms and conditional effects and uses the Causal Graph Heuristic [70] with Greedy Best-First Search.

LPG [59] deviates from this trend and does not follow the forward heuristic search approach of planning. Instead, it conducts a stochastic local search over partial plans that are subgraphs of the planning graph [12], using the Walkplan heuristic, inspired by Walksat [136], that measures constraint violations in the partial plan.

IPC 2011, 2014. Portfolio planners emerged to prominence in the 7^{th} and 8^{th} International Planning Competitions, with a total of 10 planners using the approach in IPC 2011, which increased to 29 in IPC 2014 [148]. Stone Soup [71], a portfolio with many heuristics — *blind*, $h^{\max}$, *LM-cut*, *landmarks*, and *M&S* forming the ingredients, won the *optimizing* track in IPC 2011, and IBaCOP [21] and MIPLAN [107] ranked first, and third, respectively, in *satisficing* track of the 2014. MIPLAN uses a Mixed integer program to select a suitable portfolio with the best achievable performance. On the other hand, IBaCOP uses a Pareto efficiency technique [22] to select planners.

A noteworthy approach, not using any portfolios, was that of Madagascar [126], which showed that the approach of planning as satisfiability that previously was unable to scale on large problems, could be significantly enhanced by better implementation techniques, allowing for planners that are very different from those following the approach of planning as heuristics search. Madagascar included several advances in planning as SAT, including more concise and scalable encodings, parallel/interleaved search methods, and planning heuristics for satisfiability. Madagascar is quite relevant to our work on planning as constraint satisfaction as Boolean satisfiability is a particular type of CSP, and we discuss the empirical performance of Madagascar in Chapter 5. Another successful non-portfolio approach was that of bi-directional symbolic search, used by SymBA* [145] and cGamer [144], the winners of the *optimizing* track of IPC 2014. Symbolic search represents transition functions and world states using binary decision diagrams (BDDs), potentially achieving exponential savings in memory and runtime consumption and allowing for more efficient integration of progression and regression-based search into effective bi-directional counterparts. SymBA* extends the bi-directional symbolic search

to perform A*, using parameter-based abstraction heuristics to guide the search towards opposite frontiers from both forward and backward directions.

IPC 2018, 2023. In the last two IPCs, the portfolio planners have risen from their earlier position of prominence to dominance in the ranking hierarchy, with all the winners of IPC 2023 employing some portfolio approach. Delfi [82], an online portfolio planner that uses a deep learning approach to choose the best planners for a problem, won the *optimizing* track of the IPC 2018, and Stone Soup [71], a contender of IPC 2014 as well, took the first spot in *satisficing*. Similarly, the winners of IPC 2023 [47] — Ragnarok, Decstar, FD Remix, Scorpion Maidu, and Levitron, all use sophisticated portfolio approaches to choose the correct set of algorithms, highlighting the significant interest of the community. We compare the performance of our work on optimal sequential Planning against Delfi. However, portfolio techniques can be hard to interpret. In particular, they do not offer new perspectives on solving classic planning problems, and generally, there is a limited explanation of the portfolio's performance. Hence, we do not delve into those aspects in our work.

We also observe the community acknowledging and addressing the bottleneck of *grounding* lifted representations, with more efficient grounding approaches based on more efficient general-purpose solvers like Gringo [55, 52], and lifted planning approaches that do away with grounding as a preprocessing step altogether [29]. The Powerlifted and Levitron [30] planners used the lifted planning approach, with Levitron finishing as the runner-up in the *satisificing* track, and the winner Scorpion Maidu using an efficient *grounding* approach based on Gringo. We compare this aspect in detail in the Section 4.7.

The other noteworthy approaches include Complementary, a PDB heuristic [54] based planner which used UCB1 algorithm to determine the size of the pattern database instead of sequentially increasing it over time, Scorpion, an A* planner using Cartesian Abstraction Heuristics [133] and PDBs selected by saturated cost partitioning [133]. Also, two width-based planners, BFWS-Preference and DUAL-BFWS, launch the approach into prominence, as the notion of width finally takes shape in these state-of-the-art algorithms for *satisficing* planning with the BFWS-Preference planner winning the *Agile* track and the DUAL-BFWS earning the silver medal in the *satisficing* track. We extensively study these algorithms in this research, and our work on approximations of novelty search addresses the challenges associated with the scalability of these algorithms. We submitted

our work in the IPC 2023, and an extensive evaluation of the performance of the planners is presented in Chapter 4.

Relevance to this work. For all of our experiments, we rely on the benchmark instances from the classical track in IPC 1998 to IPC 2018. These problem sets offer a broad range of instances with different dependency structures among the actions in the possible plans and a variety of features, including domain size and complexity — from small and simple ones such as Blocksworld and Gripper to large and complex ones such as Logistics, action representation — actions can have universally quantified conditions and conditional effects, plan quality — action costs specifications that challenge the search algorithms differently, plan length — from single action to ones with thousands of actions. The diversity in characteristics of the IPC instances allows us to report an unbiased comparative performance of our algorithms in this thesis. Furthermore, it enables us to understand, at a high level, how the structure of domains and properties of instances affect the planner's performance.

"If you want to walk fast, walk alone. But if you want to walk far, walk together."

Ratan Tata

3

Temporal Planning

In this chapter, we present the formalism of temporal planning and its limitations and advantages. We also survey the noteworthy computational approaches, many of which, as we see later, are incomplete, i.e., they strictly cater to a subset of temporal planning problems and may produce unsound results for temporally expressive languages.

3.1 Introduction

One of the shortcomings of STRIPS is that its semantics is defined in the context of classical *sequential* planning, which even Fikes and Nilsson point out in their 1993 retrospective [45]. Temporal planning eliminates that restriction, allowing actions to overlap and execute simultaneously. Hence, while dealing with the sequencing concern in classical planning, we must also consider concurrency in temporal planning. This changes the structure of the solution plan, too. Unlike classical planning, where we look for the best sequence of actions based on their costs, temporal planning looks for the best schedule that assigns actions to different time points. We also have another way of measuring the

quality of a temporal plan. Instead of only caring about the total cost of actions, we also care about the makespan, which is the shortest time needed to execute the whole plan. The value of this increase in expressivity is limited in inherently sequential systems, e.g., the NYT Letter Boxed and any other single-player game, which, however, becomes apparent in the case of multi-agent systems.

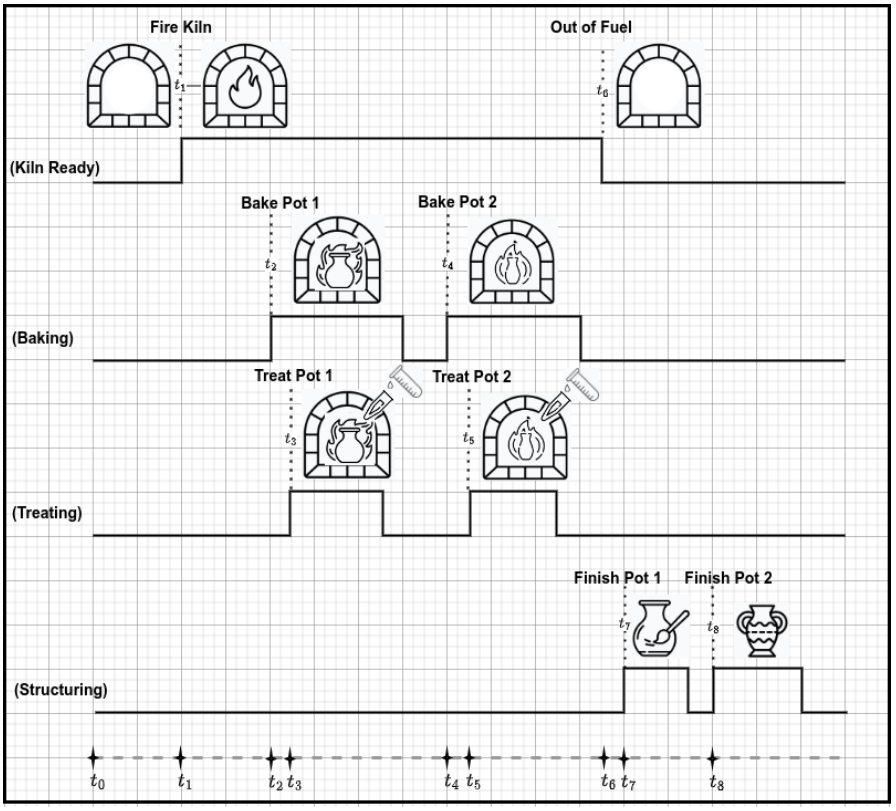


FIGURE 3.1: Chronological order of events in Temporal Machine Shop[1].

When multiple agents work together on interdependent tasks with temporal constraints, temporal planning is beneficial for finding a time-efficient solution. This type of planning problem is often encountered in robotics and industrial settings, where agents must coordinate their actions to achieve a common goal. Consider the situation of *temporal machine shop* (TMS) illustrated in Figure 3.1. The objective of the problem is to create ceramics with an efficient allocation of resources. The process of making ceramics involves four interdependent tasks or actions: *fire kiln*, *bake*, *treat*, and *finish*. The chemical treatment must be done while the pot is baking, and baking requires that the kiln is lit up. Hence, these durative actions must occur concurrently and cannot be sequenced. On the other hand, finishing the pot must be sequenced after baking and treating and is independent of whether the kiln is fired, showing that the concern about sequencing

[1]This diagram has been designed using images from Flaticon.com

persists. The TMS problem is a relatively small portion of a large system of machines with complex dependencies. An actual industry application would require more involved planning models. Yet, it cannot be expressed in the classical planning model. This highlights the importance of developing formal languages to express temporal (concurrent) planning and computational approaches to solve these problems.

Most of the initial work on computation approaches to temporal planning focused on extending classical planning, suffering from the same limitations as its parent — being unable to verify temporally expressive languages that can represent problems requiring concurrent execution. This sets the context of our work in Chapter 7, where we address this challenge of *required concurrency*.

Chapter Outline. First, the temporal planning models are explained. Then, we briefly discuss the computational complexity results. After this, we present the different computational approaches to temporal planning and the related works. Finally, we present a brief history of temporal planning from the perspective of international planning competitions.

3.2 Models of Temporal Planning

Standardizing the semantics of temporal planning has been a contentious issue, with multiple propositions suggesting different semantics. This issue percolates throughout the planners addressing temporal planning, with many assuming specific semantics according to what works for their approach. *Throughout our work, we stick to the semantics of temporal planning as specific in the work of Fox and Long PDDL2.1 [49], limiting the syntax to the STRIPS fragment — without considering the numeric effects and processes.* For this, there are two models of temporal planning, one based on dense-time representation [61] and the other based on discrete-time formulation [121].

3.2.1 Dense formulation

The standard semantics of PDDL 2.1 consider the time domain to be *dense*, i.e., between any pair of time points, there will *always* be a third. In this context, we use the model of temporal planning presented by Gigante et al. [61] to explain the semantics of PDDL

2.1. The model incorporates most of the syntactic elements of STRIPS and adds *durative actions* to the building blocks.

Definition 3.1. A **temporal planning** problem is a tuple, $P := \langle F, O, I, G \rangle$, where F is the set of propositional atoms, O is the finite set of operators, $I \subseteq F$ is the initial state, $G \subseteq F$ is the goal condition. An instantaneous operator $o \in O$ is a tuple $o := \langle \text{Pre}_o, \text{Add}_o, \text{Del}_o \rangle$, where Pre_o is a set of atoms, Add_o and Del_o are the set of *positive* and *negative* effects, respectively. On the other hand, a durative (temporal) action is a tuple $\langle o_{\vdash}, o_{\dashv}, \text{Pre}_o^{\leftrightarrow}, [L_o, U_o] \rangle$, where $o_{\vdash}, o_{\dashv}$ are instantaneous actions, $\text{Pre}_o^{\leftrightarrow}$ is the set of conditions that must hold over the duration of action and $[L_o, U_o]$ specifies the lower and upper bound on the duration of actions. When $L_o = U_o$, the duration is fixed.

Definition 3.2. A **solution** or **plan** to the temporal planning problem is a tuple $\pi := \langle (o_1, t_1, d_1), (o_2, t_2, d_2) \dots \rangle$, where o_i is a durative actions, t_i is a start time of o_i and d_i is the duration of action.

Plan Validity (Semantics). The model of temporal planning is not as closely associated with a transition system (graph) as the classical planning model. The semantics of durative actions involve conditions over multiple time points, making them harder to explain. Gigante et al. simplify this by introducing *set of time snap actions* (TSA) and *induced plan*, which allows the authors to explain plan validity in terms of classical execution of the induced plan.

Definition 3.3. A **set of time snap actions** of plan $\pi := \langle (o_1, t_1, d_1), (o_2, t_2, d_2) \dots \rangle$ is defined as the set of instantaneous actions in the plan.

$$H_\pi := \langle (t_1, o_{1\vdash}), (t_1 + d_1, o_{1\dashv}), \dots \rangle$$

Definition 3.4. An **induced plan** of plan $\pi := \langle (o_1, t_1, d_1), (o_2, t_2, d_2) \dots \rangle$ is a sequence $\pi^{ind} := \langle (\hat{t}_1, \{a \mid (\hat{t}_1, a) \in H_\pi\}), \dots \rangle$, ordered in terms of increasing time values, i.e., $\forall i < j,\ \hat{t}_i < \hat{t}_j$.

The *induced* plan consists of a set of time snap actions executing simultaneously at the same time point, but it says nothing about the structure of these actions. This is important as the snap actions may contain mutually exclusive atoms in their preconditions and effects that cannot hold together. Hence, the existence of *mutex* actions would invalidate the plan.

Definition 3.5. A **pair of snap actions** (a, a') **are mutex** iff $\text{Pre}_a \cap \text{Del}_{a'} \neq \phi$, or $\text{Pre}_{a'} \cap \text{Del}_a \neq \phi$, or $\text{Add}_{a'} \cap \text{Del}_a \neq \phi$, or $\text{Add}_{a'} \cap \text{Del}_a \neq \phi$.

Definition 3.6. A **plan** $\pi := \langle (o_1, t_1, d_1), \dots (o_n, t_n, d_n) \rangle$ **is valid** if the induced plan π^{ind} maps into goal when executed sequentially, the overall and durative constraints hold, and any pair of mutually exclusive instantaneous actions do not occur at the same time. That is, the following conditions are met.

1. The action durations are within limits, $L_{a_i} \leqslant d_i \leqslant U_{a_i}$
2. Mutex actions do not occur at the same time point, $\forall I \in \{1, \dots, m\}, \nexists (a, a') \in B_i, mutex(a, a')$, where $\pi^{ind} := \langle (\hat{t}_1, B_1), \dots (\hat{t}_m, B_m) \rangle$
3. For all the actions in any set B_i of the induced plan $\pi^{ind} := \langle (\hat{t}_1, B_1), \dots (\hat{t}_m, B_m) \rangle$, the preconditions hold in the previous state $\bigcup_{a \in B_i} \text{Pre}_a \subseteq s_{i-1}$, where s_i is defined as $(s_{i-1} \backslash \bigcup_{a \in B_i} \text{Del}_a \cup \bigcup_{a \in B_i} \text{Add}_a)$ and $s_0 = I$
4. The final state achieves the goal, $G \subseteq s_m$
5. The overall preconditions hold over the duration of the action, i.e., $\forall (o, t, d) \in \pi$, $\forall k \in \{1, \dots, m\}$, $\forall \hat{a} \in B_k$, $t \leqslant \hat{t}_k \leqslant t + d \rightarrow \text{Pre}_o^{\leftrightarrow} \subseteq s_k$, where $\langle (\hat{t}_1, B_1), \dots, (\hat{t}_m, B_m) \rangle$ is the induced plan

ϵ-separation. Mutually exclusive (mutex) events must be separated by a non-zero amount of time. Historically, there has been ambiguity in the semantics of time difference in this context. Gigante et al. [61] clarify this by separating it into two classes. One follows the *ϵ-separation* semantics where the amount of time separation ϵ is given upfront, and otherwise, the semantics is that of > 0-*separation*.

Complexity of temporal planning with dense-time semantics. Gigante et al. [61] prove that the complexity of the temporal planning model in Definition. 3.1 with ϵ-separation is EXPSPACE-complete when self-overlapping of actions is allowed, otherwise it is PSPACE-complete.

3.2.2 Discrete formulation.

Rintanen [121] presents a different temporal planning model based on discrete-time representation. Here, the time domain is $\mathbb{N} = \{0, 1, \dots\}$, and the action preconditions and effects are defined over the set of time points relative to the start time.

Definition 3.7. A **discrete-time temporal planning** model is defined as a tuple $P = \langle A, I, O, G, R, D \rangle$ where, A is the finite set of state variables, O is the finite set of actions, $I : A \to \{0, 1\}$ is the initial state (a *valuation* of A), $G \in \mathcal{L}$ is the goal, $R : O \to \mathcal{L}$, maps actions to their preconditions and D is a finite set of rules.

In this, $\mathcal{L}$ is a temporal language with operators $[i \dots j]\phi$, $i, j \in \{\dots, -1, 0, 1, \dots\}, i \leqslant j$ is used to define durative conditions, $[i \dots j]\phi$ means that ϕ holds in all time points $\{i + t, \dots, j + t\}$, where t is the current time point. Also, *rules* are used to define *conditional* effects, where a *rule* is a tuple $\langle p, e \rangle$ where $p = [i, \dots, j]\phi$ is in $\mathcal{L}$ and e is a set of atoms, implying that if ϕ holds at points $\{i + t, j + t\}$, then $l \in e$ is made *true* at $t + 1$.

The semantics of the model involves defining a *valuation* function $v : N \times (A \cup O) \to \{0, 1\}$ that captures whether an atom holds or an action is taken at a time point. To establish the relationship between the actions in the plan and the *valuation function*, a relational predicate $\models_t$ is defined

$v \models_t a$ iff $t \geqslant 0 \wedge v(t, a) = 1$ or $t < 0, v(0, a) = 1$,

$v \models_t o$ iff $v(t, o) = 1$,

$v \models_t \neg\phi$ iff $v \not\models_t \phi$,

$v \models \phi \wedge \phi'$ iff $v \models \phi \ \wedge \ v \models \phi'$,

$v \models \phi \vee \phi'$ iff $v \models \phi \ \vee \ v \models \phi'$,

$v \models_t [i, \dots, j]\phi$ iff $\forall h \in \{i, \dots, j\}, v \models_{t+h}$

where $a \in A$, $o \in O$, $\phi, \phi' \in \mathcal{L}$, and $i \leqslant j$.

Definition 3.8. A **plan** $\pi : T \to 2^O$ **is valid** iff there exists a *valuation* such that

1. $v(0, a) = I(a)$
2. Actions in the plan are true in the *valuation*, $v(i, o) = 1$ iff $o \in \pi(i)$,
3. All conditional effects are applied, $v(i, a) = 1$ if $\langle p, e \rangle \in D$, $a \in e$, $v \models_{i-1} p$, and $v(i, a) = 0$ if $\langle p, e \rangle \in D$, $\neg a \in e$, $v \models_{i-1} p$

4. Frame axioms hold, $v(i, a) = v(i-1, a)$ if there is *no* rule p with effect literals a or $\neg a$ such that $v \models_{i-1} p$,

5. Preconditions of actions in the plan are satisfied, $v \models_i R(o)$ for all $o \in \pi(i)$,

6. Mutex effects are separated, i.e., no pair of rules $(\langle p, e\rangle, \langle p', e'\rangle)$ in $D \times D$ exist such that $a \in e$ and $\neg a \in e'$ and $v \models_i p \wedge p'$

7. Goal condition is met, $t = \max(T)$, $v \models_t G$

Complexity of temporal planning with discrete-time semantics. Rintanen [121] proves that the complexity of temporal planning model in Definition. 3.7 is EXPSPACE-complete.

3.3 Computational Approaches

3.3.1 Search and scheduling

Most common approaches to temporal planning fall under the bracket of techniques that look at it as an intersection of problems of searching for actions in the plan and scheduling them such that the temporal constraints hold. The search component typically borrows methods from the classical planning counterpart, including *heuristics* repurposed for concurrent planning and mutex computations. This section explains the typical variations in how the search and scheduling problems are framed and solved.

STRIPS reduction and rescheduling. Early temporal planners, including LPG [59], MIPS [37], and YAHSP-MT [151], used the simple approach of compiling the temporal planning problem into a sequential one and solving it with a classical planning approach, and then greedily rescheduling the actions in the plan to obtain a temporal plan. *Simple* does not mean that these approaches do not perform well. The YAHSP won the temporal track in IPCs of 2011 and 2014, and LPG and MIPS were winners of the first-ever temporal track, IPC 2002. However, all of them have two drawbacks. One, they are *incomplete*, i.e., they do not cover all the instances with *required concurrency*, and two, they cannot be used for *optimizing makespan*.

Decision Epoch Planning. Another approach is choosing a timestamp *first* and then deciding the actions to execute at the timestamp, i.e., the search progresses by choosing

a new action at the current timestamp or progressing the timestamp. However, this method has similar drawbacks of inability to handle *required concurrency* cases [31]. Many planners, including TLPLAN[3], TFD [42], and SAPA [36] follow this approach.

Forward-chaining. Planners including CRIKEY [64, 27], COLIN [26], POPF [25], and OPTIC [8] follow an approach that involves making two different kinds of decisions during the search — while the plan is being constructed, differentiating it from the trivial method of STRIPS reduction and rescheduling. The first decision is what action to add to the plan, and the other is when to execute it. While CRIKEY and COLIN planners perform a total-ordered search, POPF and OPTIC adopt a forward-chaining strategy that allows for partial-ordering of actions by delaying the commitment to order. The temporal constraints are handled by constructing a Simple Temporal Problem (STP) [32] that can be solved in polynomial time by solving the shortest-path problem within the directed graph representation of the STP, a Simple Temporal Network (STN).

The forward-chaining planners are suggested to be *complete*, i.e., they can handle *temporally expressive* planning languages and have state-of-the-art performance in *optimizing* planning. However, as we will explain in Chapter 7, the approach seems to struggle with some trivial problems with required concurrency.

Decomposition-based approaches. In contrast to the above methods, TGP [140, 96] and ITSAT [118] follow a path of one-shot decompositions, in which there is no iteration.

3.3.2 Direct compilations into CSPs

The components of temporal planning, including numeric state variables, durations, intervals, and scheduling constraints, all have the flavor of a CSP. Hence, it is surprising that there are not as many works in this direction, highlighting the difficulty of formally expressing the semantics of temporal planning, including the *temporally expressive* languages. A prominent work demonstrating compilation into CP is CPT [152] that models the restricted case [140] of temporal planning. It has directly inspired our work on encoding planning into CP for both temporal and sequential planning. We explain the encoding of CPT in more detail in Chapter 7. The ITSAT [118] planner is another candidate in this space. It compiles temporal planning into Boolean Satisfiability (SAT) — letting the SAT solver handle the logical component, and solves an STP to verify the

STN. More recently, Panjkovic [109] presented encodings of temporally expressive planning into Optimal Modulo Theory (OMT) that can handle both required concurrency and makespan optimal planning.

3.4 Temporal Planning in International Planning Competitions

The first competition to focus on temporal and metric planning was the 3^{rd} planning competition, which was organized in 2002 under the chairmanship of Maria Fox and Derek Long. The planners were required to deal with more complex domains and problems involving time and numbers (such as scheduling and resources), and therefore, the competition used the newly proposed language of PDDL 2.1 [48] that builds upon the syntax of PDDL 1.2 [60], additionally allowing for modeling of temporal and numeric aspects. Two fully automated planners, LPG [59] and MIPS [37], showed distinguished performance of the temporal planning instances, both of which use an approach to reschedule total order plans and could handle *temporally simple* planning problems. The 4^{th} planning competition, held in 2004, saw the rise of planning as constraint satisfaction as a performant approach for temporal planning, with a reduction of *temporally simple* planning into constraint programming, CPT [152] winning the *optimal* track. CPT extends the CHOCO CP library [88] with more efficient implementations of certain propagators and does a lot of reasoning in the preprocessing stage, including reachability using h^m heuristic, to propagate domain constraints on fluent and action variables.

The temporal track, especially the *optimal temporal planning* track, has received lackluster attention since the 5^{th} international planning competition, IPC 2006, which saw CPT winning the temporal optimizing track again. The *optimizing* track was canceled in IPCs held in 2008, 2014, and 2018 due to a lack of submissions, and the temporal track was entirely absent from the most recent IPC 2023, highlighting a lack of progress in the field. The temporal *satisficing* track continued to receive fair attention until IPC 2018, in which the noteworthy planners were Temporal Fast Downward (TFD) [42], YAHSP-MT [150, 151], and POPF [25, 8]. TFD is an extension of the Fast Downward planning system to temporal planning, which implements the *decision-epoch* approach and uses context-sensitive additive heuristics h^{cea} [70] to guide the search. YAHSP-MT uses the same rescheduling approach as LPG and MIPS. POPF comes from the family

of planners starting with CRIKEY [64], in which the problem is decomposed into the problem of finding a partial ordering of actions, which could be done by *lifting* a total order plan, and the temporal component of when to execute them that is managed using a simple temporal network (STN).

Part II

Tractable Approximations of Width-based search

"Nothing in life is to be feared, it is only to be understood. Now is the time to understand more, so that we may fear less."

Marie Curie

4

Approximate Novelty Search

Width-based search algorithms seek plans by prioritizing states according to a suitably defined measure of Novelty that maps states into a set of Novelty categories. Space and time complexity to evaluate state Novelty is known to be exponential on the cardinality of the set. We present novel methods to obtain polynomial approximations of Novelty and width-based search. First, we approximate Novelty computation via random sampling and Bloom filters, reducing the runtime and memory footprint. Second, we approximate the best-first search using an adaptive policy that decides whether to forgo the expansion of nodes in the open list. These two techniques are integrated into existing width-based algorithms, resulting in new planners that perform significantly better than other state-of-the-art planners over benchmarks from the International Planning Competitions.

4.1 Introduction

Autonomous systems operating on the edge of computer networks or that only have occasional, sporadic access to vast, centralized computing resources require decision-making algorithms that work under those conditions. Not only are low response times required to seek courses of action that are safe and effective, but also, the memory available for such computation is limited. The need to adapt existing heuristic search algorithms, such as A*, to deal with time and space restrictions was recognized early on [23] and followed-up recently [146, 34].

In this chapter, we look at *width-based search methods* [91], a family of algorithms that rely on heuristics that measure the *novelty* of a state, comparing its information content with that of states visited in the past. Originally developed in the context of classical planning [57], when combined with other heuristics [15, 72, 93, 81], width-based planners become state-of-the-art and competitive with portfolio solvers [116]. A major shortcoming of the latter width-based methods derived from best-first search (BFS) [38, 110], such as those used by the planner DUAL-BFWS, is that measuring Novelty is exponential on the number of discrete levels or categories used to rank states. Lipovetzky and Geffner [91] showed that an upper bound exists for any given classical planning instance, yet this result cannot be exploited, as this bound results in impractical runtimes to evaluate states, a crucial issue for the effectiveness of heuristic search methods.

We address this issue by proposing new methods to obtain polynomial approximations of Novelty and to control the growth of the memory footprint of BFWS algorithms. The first contribution is an appraisal approximation of the Novelty of state information by randomly sampling the space of possible valuations of state variables and using Bloom filters for efficient but imprecise storage of state information. The second contribution is a novel form of best-first search, which uses an adaptive policy that decides whether to delay the generation of successor states. This policy is derived from the analytical solution to an infinite-horizon Markov Decision Problem (MDP) [9], where its cost function controls the representation of different Novelty categories in the open list.

Chapter Outline The chapter is structured as follows. We concisely discuss background material covering classical planning, width-based search, and Bloom filters. Sections 4.3 and 4.4 expound the contributions of this work, approximations of Novelty measurements,

and search. We evaluate both over every benchmark in the IPC satisficing track. We finalize with a discussion of our results' importance and potential impact.

4.2 Understanding Novelty Measure in Classical Planning

The planners discussed in this chapter are instances of the classic heuristic search algorithm named Best-First Search [38, 110] or BFS for short. This algorithm searches for plans by incrementally extending all paths (nodes) in $S(P)$ starting from s_0. The nodes are visited in the order specified by an *evaluation function* f defined over paths, and the algorithm terminates in the first path that ends on a state $s \in S_G$. BFS implicitly enumerates the states s in $S(P)$ by assigning to them a natural number, the *expansion order*, $e(s)$ [34], which is 0 for s_0 and increases by one unit for every new state s extending an existing path. We denote the set of states generated before s as $\mathcal{P}(s) = \{\, s' \,|\, e(s') < e(s)\}$. As we will see in the next section, this ordering is crucial to define *Novelty measures*.

4.2.1 Best-First Width Search (BFWS)

BFWS [93] is a family of BFS algorithms where the evaluation function for a node n, $f(n)$ is defined as a tuple of functions

$$f(n) = (w, h_1, \ldots, h_m)$$

where $w : S \to \mathcal{W}$ is the function measuring novelty, that maps states $s \in S$ into categories $\omega \in \mathcal{W}$, $\mathcal{W} \subset \mathbb{N}$, and $H = \{h_1, \ldots, h_m\}$ is a set of suitably chosen functions. When inserting nodes n in the open list, BFWS algorithms sort the list in increasing order according to the first function in $f(n)$, breaking ties recursively with the provided h_i. These functions can also be used to partition the set of states generated before s as $\mathcal{P}(s, H) = \{\, s' \,|\, e(s') < e(s), h(s) = h(s')\ \forall h \in H\}$.

Definition 4.1. The *novelty* $w(s) = w_{\langle H \rangle}(s)$ of a state s given a set of partition functions H over states $s \in S$ is k, iff (1) exists a tuple[1] $t \subseteq F$ of size k, s.t. $s \models t \wedge \forall s' \in \mathcal{P}(s, H)$, $s' \not\models t$, (2) for all tuples $t \subseteq F$ of size $k' < k$, $s \not\models t \vee \exists s' \in \mathcal{P}(s, H)$, $s' \models t$.

[1] Conjunction of atoms.

As the introduction notes, BFWS algorithms are state-of-the-art compared to the IPC classical planning benchmarks. We explain the BFWS framework by discussing one of the best-performing algorithms in the Agile track (IPC 2018), BFWS(f_5), in detail.

The evaluation function $f_5 = \langle w, \#g \rangle$ makes BFWS expand novel states first, breaking ties with a simple goal counting heuristic [106] $\#g(s)$. The Novelty function uses two heuristic functions to partition the Novelty space $w = w_{\langle \#g, \#r \rangle}$, one is goal count, $\#g(s)$, and the other, $\#r(s)$, counts the atoms p achieved along the path to s, such that $p \in R$, $R \subseteq F$, where R is selected by a *relevance analysis* procedure. R is meant to contain atoms which are *instrumental* to reach the goal efficiently, so for domain-independent planning, one can instantiate R as a set of landmarks [72], or the set of fluents which belong to positive effects of actions in the relaxed plan from s_0 [73]. Both of the above definitions of R were used in the planner DUAL-BFWS, and the latter was used in BFWS(f_5).

Evaluating $w(s)$ requires testing states s to belong to the categories $\omega \in \mathcal{W}$. Lipovetzky and Geffner [93] define $\mathcal{W}$ as the integer interval $[1, i+1]$ where i is the size of the largest novel tuple generated by optimal plans for P. In this case, the test above requires to generate exhaustively all tuples t of size i present in state s and determine if they are present in $\mathcal{P}(s, H)$. An optimal procedure to implement Definition 4.1 follows. For each tuple of size $1 \leqslant l \leqslant i$, let $\beta_l(s) = \{t \mid t \subseteq F,\ s \vDash t,\ |t| = l\}$[2]. Starting with $l = 1$, we enumerate tuples $t \in \beta_l(s)$, and then test if t is part of previously observed tuples in the set $\mathcal{N}(s) = \bigcup_{s' \in \mathcal{P}(s,H)} \beta_l(s')$.

If the test is negative for at least one tuple, then $w(s) := l$. Otherwise, we must test the elements of β_{l+1} until $l = i$. If the test is positive for all $t \in \beta_i$, then the state is considered not novel, and $w(s) := i + 1$. This leads to the exponential time and space $O(|F|^i)$ requirements to evaluate $w(s)$ that we outlined in Section 4.1. While i is known to be generally way smaller than $|F|$, the bound i is usually high enough to render the Novelty test up to i impractical. In the case of the IPC benchmarks, any value of $i > 2$ leads to very high runtimes to generate states. In practice, $w(s)$ is approximated by setting i to an arbitrary lower bound, which renders the evaluation of $w(s)$ to be tractable but may relegate states with valuable information to the back of the open list.

[2]We note that β_l can be iterated by lazily generating its elements.

4.2.2 Bloom Filters

Bloom filters [11, 95] are a probabilistic data structure to represent sets efficiently, at the expense of allowing false positives when testing whether the set contains a given object. Typical implementations of Bloom filters consist of a bit-array v of size r, where all entries v_j are initially set to $\bot$, and K independent hash functions η that map objects into the range $[1, r]$. To add an object o as a member of the set represented by the Bloom filter, the K hash functions η_l are evaluated on o, so $v_{\eta_l(o)} \coloneqq \top$ for $l = 1,..., K$. To test whether o is in the set, the hash functions η_l are evaluated, and if all $v_{\eta_l(o)} \coloneqq \top$, then o is considered to be an element of the set. Compared to a traditional hash table whose size grows with that of the range of possible objects, the Bloom filter has fixed-size r.

The value choice for r determines the probability of obtaining a *false positive*. That is, testing o for containment and getting a positive answer when o has not been previously added as a member. As noted in [18], the probability of a false positive is given by

$$P_f = \left(1 - e^{-\frac{Kq}{r}}\right)^K \tag{4.1}$$

where q is the *expected* number of different objects to be tested. The analytical solution to the problem of minimization of false positive rate concerning K shows that P_f is minimized when $K = (r/q)\ \ln 2$ [18]. Since the expected number of different objects, nodes in planning, tends to be larger than the memory, it follows that when $(r\ \ln 2) \leqslant q$, then $K = 1$ minimizes P_f.

4.3 Novelty Approximation

In this Section, we describe an approximate measure of Novelty for newly generated states, $\hat{w}(s)$, which is tractable and can be proved equal to $w(s)$ with positive probability. For that, Definition 4.1 is relaxed as follows.

Definition 4.2. The *approximate novelty* $\hat{w}(s) = \hat{w}_{\langle H \rangle}(s)$ of a state s given a set of partition functions H over states $s \in S$ is k, iff (1) exists a tuple $t \subseteq F$ of size k, s.t. $t \in Z_k(s) \wedge \forall s' \in \mathcal{P}(s, H),\ \mathcal{O}(s', t) = \bot$, (2) for all tuples $t \in Z_{k'}(s)$ of size $k' < k$, $\exists s' \in \mathcal{P}(s, H),\ \mathcal{O}(s', t) = \top$.

We have changed Definition 4.1 in two ways. First, we only test the tuples from a randomly sampled set $Z_l(s) \subseteq \beta_l(s)$, for $l = 1,...,i$. We require that the probability of every tuple in $\beta_l(s)$ being selected is uniformly distributed. For that we sample *without replacement* from the *discrete uniform distribution* over $\beta_l(s)$, with probability mass function $p_X(t) = z/|\beta_l(s)|$ for all $t \in \beta_l(s)$, $z = |Z_l(s)|$, representing the probability of occurrence of t in $Z_l(s)$. Second, we replace the condition $s' \not\models t$ for a random variable $\mathcal{O}(s', t)$ that models the runtime behavior of a Bloom filter. $\mathcal{O}(s', t)$ maps pairs s', t to $\top$ with probability 1 when $s' \models t$ and $t \in Z_l(s')$, otherwise, it maps s', t to $\bot$ with positive probability. The tractability of $\hat{w}(s)$ follows from requiring r, the number of entries in the Bloom filter, and $|Z_l(s)| \leqslant \bar{Z}$, where $\bar{Z}$ is the maximum size of $Z_l(s)$, to be constants, e.g., $\bar{Z} = r = |F|$. It is trivial to note that the running time of any reasonable algorithm for computing $\hat{w}_{\langle H \rangle}(s)$ as per Definition 4.2 is $O(i\bar{Z})$ and memory requirement is $O(ir)$, dropping the complexity of tuple membership checks from exponential to linear on i.

These two simple changes suffice to allow measures of Novelty that are much finer than what can be obtained with highly optimized implementations of $w(s)$, but certainly, and as noted at the beginning of the Section, there is a certain probability that $\hat{w}(s)$ and $w(s)$ will not be in agreement. The rest of this Section is devoted to providing a probabilistic model of the rate at which approximate and actual Novelty disagrees.

4.3.1 Impact of Sampling

We proceed now to derive the probability of error induced by sampling from $\beta_l(s)$ following the discrete uniform distribution. By *error* we refer to the event of $w(s) \neq \hat{w}(s)$, that is $\hat{w}(s)$ is greater or less than $w(s)$ for a state s. We start by defining the probability γ_t, given s and $\mathcal{P}(s, H)$, of a particular tuple $t \in \beta_l(s)$ observed as *new* in s, that is $t \notin \bigcup_{s' \in \mathcal{P}(s,H)} Z_l(s')$ and $t \in Z_l(s)$, as

$$\gamma_t = \left(\prod_{s' \in \mathcal{P}'_t(s)} \left(1 - \frac{z}{|\beta_l(s')|} \right) \right) \frac{z}{|\beta_l(s)|}, \tag{4.2}$$

where $\mathcal{P}'_t(s)$ is defined as the set $\{s' \mid s' \in \mathcal{P}(s, H), s' \models t\}$, z is the sample size, $z/|\beta_l(s)|$ follows from the probability mass function p_X. The event of taking a sample at s is *independent* from that of sample at $s' \in \mathcal{P}(s, H)$, which allows us to use the *product rule*. Also, it follows that as $z \to |\beta_l|$ the probability $\gamma_t \to 0$, when $\mathcal{P}'_t(s) \neq \varnothing$, and $\gamma_t \to 1$,

when $\mathcal{P}'_t(s)=\varnothing$. That is, if all tuples $t \in \beta_l(s')$ are sampled in each $s' \in S$, as we do when computing Novelty exactly, the probability of t being *new* in s is 0, if $t \in \bigcup_{s' \in \mathcal{P}(s,H)} Z_l(s')$, and 1 otherwise.

From Equation 4.2, we follow that the probability of tuple t *not* being *new* is $[1-\gamma_t]$. We use this result to compute the probability that none of the tuples $t \in \beta_l(s)$ are *new*, assuming *independence* between different tuples to make the derivation tractable, as

$$p_l = \prod_{t \in \beta_l(s)} [1-\gamma_t]\,. \tag{4.3}$$

Using Equation 4.3, we can now define the probability of approximate Novelty measure to be greater or smaller than actual, $P_H = P(\hat{w}(s) > w(s))$, $P_L = P(\hat{w}(s) < w(s))$ respectively, as

$$P_H = \prod_{i=1}^{w(s)} p_i, \quad P_L = \left(1 - \prod_{i=1}^{w(s)-1} p_i\right). \tag{4.4}$$

Finally, the probability of approximate and actual Novelty measures to agree, $P_C = P(\hat{w}(s) = w(s))$, is :

$$P_C = \left(\prod_{i=1}^{w(s)-1} p_i\right)\left(1 - p_{w(s)}\right). \tag{4.5}$$

In Lemma 4.3 we prove the sum of probabilities in Eqs. 4.4 and 4.5 to be 1, and where $(P_L + P_H)$ is the total probability of sampling-induced error.

Lemma 4.3. *The sum of probabilities P_L, P_H and P_C is* 1.

Proof. Let $q = \prod_{i=1}^{w(s)-1} p_i$ be the probability of not finding new tuples with Novelty below $w(s)$. Then, Equations are rewritten as $P_L = 1-q$, $P_H = q\,p_{w(s)}$ and $P_C = q\,(1-p_{w(s)})$ for a given Novelty $w(s)$. Hence, $P_L + P_H + P_C = (1-q) + q\,p_{w(s)} + q\,(1-p_{w(s)}) = 1 - q + q\,p_{w(s)} + q - q\,p_{w(s)} = 1$, proving the correctness of equations. □

4.3.2 Synergies between Sampling and Bloom Filters

$\mathcal{P}(s)$ size is $O(b^d)$, that is, exponential on the branching factor b of the transition system $S(P)$ and the length d of the path to s from s_0. Therefore, replacing $\mathcal{N}(s) = \bigcup_{s' \in \mathcal{P}(s,H)} \beta_l(s')$ by a Bloom filter with r entries cannot come for free. While the Bloom filter will always give the correct answer to membership queries for tuples t that are in $\mathcal{N}(s)$, it can produce false positives for membership, as it incorrectly gives a positive answer for tuples $t' \notin \mathcal{N}(s)$.

We note that sampling from β_l enables the use of Bloom filters to "approximate" $\mathcal{N}(s)$. This is because it leads to a reduction of the probability of false positives P_f given in Equation 4.1, in comparison with what we would obtain from using β_l directly as in the algorithm for $w(s)$ given in Section 4.2.1. This observation follows from noting that q in Equation 4.1 is the expected number of distinct tuples t sampled during the search, and the rate of growth of this random variable is directly proportional to $\bar{Z}$. The smaller $\bar{Z}$ is, the slower q will grow. From Equation 4.1, we can see that the probability of false positives P_f increases with the ratio q/r, which in turn depends only on q as r is a constant. Therefore, the rate of growth of P_f depends on $\bar{Z}$.

Finally, we note that P_f will be maximized when q is exponential on l, the maximum size of the tuples considered. Then, in principle, the false positive probability increases, too as l grows larger.

4.3.3 Total probability of error.

To obtain the total probability of erroneously appraising the Novelty of a newly generated state s, P_{error}, we incorporate Equation 4.1 into Equation 4.2

$$\gamma_t = \left[\left(\prod_{s' \in \mathcal{P}'(s)} \left(1 - \frac{z}{|\beta_l(s')|} \right) \right) \frac{z}{|\beta_l(s)|} \, (1 - P_f) \right] \tag{4.6}$$

used to evaluate Equation 4.4, from which then it follows that $P_{error} = P_L + P_H$.

4.3.4 Conjoining BFWS(f_5) and Novelty Approximation.

Novelty approximation using sampling and Bloom Filter can be directly applied to BFWS. We replace the $w(s)$ in BFWS(f_5) with the approximation $\hat{w}(s)$ resulting in $\hat{f}_5 = \langle \hat{w}, \#g \rangle$.

Additionally, any reasonable implementation to compute $\hat{w}_{\langle \#g, \#r \rangle}(s)$ for BFWS($\hat{f}_5$), as per Definition 4.2, needs to track the evaluations of partition functions $H = \{\#g, \#r\}$ for all observed tuples. This increases the space complexity by a factor of the number of possible partitions, $|G| \times |F|$. We manage the increase in space complexity by employing a set of Bloom filters, a *bank* V, and then bounding the space available for Novelty computation by a parameter D_{max}. In case D_{max} is sufficiently large to track all the tuples and evaluations of partition functions, we enable exact item membership tests. Otherwise, we use the *bank* of Bloom filters. Whenever a new partitioned space is observed, we assign it a Bloom filter from V. If the number of observed partitions exceeds $|V|$, we overlap them randomly, allowing different partitions to use the same Bloom filter. This results in a gradual decrease in the accuracy of Novelty computation in exchange for space; the P_{error} increases as more partitions overlap.

The resulting planner BFWS($\hat{f}_5$) has the following hyperparameters, namely, the sample size $\bar{Z}$, the size of a Bloom filter r and the bound of space D_{max}. In Section 4.5, we present experimental evaluations with different choices of parameter values.

4.3.5 Increasing the Novelty bound

As discussed in Section 4.2.1, any implementation of Definition 4.1 has a complexity of $O(|F|^i)$, rendering the computation impractical for many instances. Whereas, Definition 4.2 has linear complexity allowing us to compute $\hat{w}(s)$ for any value of $i \in [1, |F|]$. In the following section, we describe the impact of increasing $\mathcal{W} \in [1, i+1]$ in the *polynomial* planners: BFWS(f_5) with Novelty pruning [92].

Theorem 4.4. *Let $P = \langle F, O, I, G \rangle$ be a STRIPS planning problem. The number of nodes generated for each Novelty category $\omega \in \mathcal{W}$, when run with P as input, is less than or equal to $\binom{|F|}{\omega} \times |G| \times |F|$.*

Proof. The upper bound on the number of observed state partitions is given by $|G| \times |F|$. Also, the count of tuples of size ω, of atoms in F, is $\binom{|F|}{\omega}$. Hence, the number of nodes with $w(s) = \omega$ cannot exceed $\binom{|F|}{\omega} \times |G| \times |F|$. □

Corollary 4.5. *Let $P = \langle F, O, I, G \rangle$ be a STRIPS planning problem, and BFWS($\hat{f}_5$) the polynomial planner using Bloom filters introduced above. When run with P as its input, BFWS($\hat{f}_5$) generates at most $|V| \times r$ nodes for each Novelty category $\omega \in \mathcal{W}$.*

Proof. A Bloom filter represents $\binom{|F|}{\omega}$ tuples, but the number of true negatives is bound by the size of Bloom filter r. Also, we create a set of Bloom filters, the *bank* V, that represents the set of partitioned spaces of cardinality $|G| \times |F|$. Hence, the above bound holds. □

From Theorem 4.4, we note that the bound on number of nodes with $w(s) = \omega + 1$ increases by $O(|F|)$ in comparison to those with $w(s) = \omega$, which makes nodes with large value of $\hat{w}(s)$ unlikely candidates for expansion. This leads us to another critical issue afflicting BFWS algorithms, inherited from BFS, that is only a small fraction of the nodes that make it into the open list are ever considered for expansion. We can use partial expansions [154] in BFWS(f_5), only adding the successor s' of s to the open list if $\hat{w}(s')$ $\leqslant \hat{w}(s)$. While this reduces the space consumed by the open list, it does not make the expansion of nodes with large $\hat{w}(s)$ value more likely, which is a key motivation behind using Novelty approximation. In fact, *polynomial* BFWS(f_5) uses similar methodology, where a successor s' of s is added to the open list only if $\hat{w}(s') \leqslant \bar{\omega}$, where $\bar{\omega}$ is an arbitrary lower bound in i.

Another method to address the above challenge is to choose a small size for r and V, and from Corollary 4.5 we can deduce that it will bound the nodes in each Novelty category by $|V| \times r$, which makes it seem that nodes with high $w(s)$ are now more likely to expand. However, in practice, the same set of nodes receive a higher value of $\hat{w}(s)$ because of an increase in P_f, which is undesirable as it does not result in more problems solved — novelty approximation is expected to allow the search to explore states of higher novelty value that are necessary to reach the goal in some instances. In the following section, we discuss a method that remediates this.

4.4 Best-First Search with Open List Control

We propose an example of a novel methodology to design BFS algorithms that aim at controlling the rate of growth of nodes of each category $\omega \in \mathcal{W}$ in the open list. To do this, we model the search as a discrete-time dynamical system subject to perturbation, and an optimal control problem [9] is formulated where optimal policies ensure that a rate of growth less than the branching factor b of $S(P)$ is sustained. With some simplifying assumptions, the optimal policy for this control problem can be derived analytically, as shown below, and integrated directly into the search algorithm. We model the evolution over time of the internal state (i.e., size of open lists) of a BFWS-like algorithm B, subject to function T_B, abstracting the instructions executed in one iteration of the expansion loop of B, as the dynamical system

$$x_{k+1} = T_B(x_k, u_k, c_k),$$

where k is the index of the current expansion, x_k is a suitably defined abstraction of the internal state of the search algorithm B at time k, $u_k \in [0,1)^{|\mathcal{W}|}$ is the *control action*, that prescribes the *pruning rate* for states with $w(s) = \omega$, and $c_k \in \mathbb{N}^{|\mathcal{W}|}$ is the count of successor states s' at time k with $w(s') = \omega$. c_k is the *perturbation*, that is, an uncontrollable side-effect of node expansion that has been modeled as a uniformly distributed discrete random variable.

The information we track in x_k is given by the tuple $\langle n_e, n_v(\omega)\rangle$, where n_e is the number of expanded nodes so far, and $n_v(\omega)$ is the count of *novel states* visited for each Novelty category $\omega \in \mathcal{W}$. If $u = 0^{|\mathcal{W}|}$, then T_B is deterministic and B behaves like a standard BFWS algorithm. Otherwise, the successors s' of state s pointed at by node n in the open list with $\min f(n)$ are generated with probability $1 - u^{\omega}$ when $w(s') = \omega$. If some s' is pruned, n is kept in a *holding queue* and re-expanded whenever the open list becomes empty. A vital implementation detail for B is that it needs to maintain $|\mathcal{W}|$ open lists in parallel, as keeping smaller open lists is often more performant [19], also greatly facilitating implementation and computation of states x_k.

To formulate an optimal control problem, we must first specify a cost function. Considering a very large number of stages or expansions, we can reasonably assume that the horizon is infinite and define the average cost per stage function [9], with a policy

$\pi = \{\mu_0, \mu_1, ...\}$, of the form

$$J_\pi = \lim_{N\to\infty} \frac{1}{N} \mathop{\mathbf{E}}_{c_k} \left[\sum_{k=0}^{N-1} g(x_k, \mu_k(x_k), c_k) \right] \tag{4.7}$$

The choice of the cost per stage $g(x_k,\ \mu_k(x_k),\ c_k)$ is dictated by the need to seek a trade-off between the number of states in the open list with $\hat{w}(s) = \omega$ growing too large and missing out useful novel states. We define g_k as follows

$$g_k = \sum_{\omega\in\mathcal{W}} \left[c_k^\omega\ (1 - \mu_k^\omega(x_k)) + \frac{1}{(1 - \mu_k^\omega(x_k))} \right] \tag{4.8}$$

where the expected count of successor nodes, with Novelty $\hat{w}(s) = \omega$, added to the open list at time k is given by $c_k^\omega\,(1 - \mu_k^\omega(x_k))$, and the second term is the inverse rate of node generation, as we want every possible value of Novelty to be represented in the open list with positive probability. Using Equation 4.8 we can rewrite J_π as

$$J_\pi = \lim_{N\to\infty} \frac{1}{N} \mathop{\mathbf{E}}_{c_k} \left[\sum_{k=0}^{N-1} g_k \right] \tag{4.9}$$

We make an assumption to facilitate obtaining the optimal policy, namely, μ_k will converge to some stationary μ as $k \to \infty$, so it can be used to estimate the cost of future stages accurately. Also, the expected value of uniformly distributed random variable c_k^ω is calculated from $n_v(\omega)$ and n_e, as $E\left[c_k^\omega\right] = n_v(\omega)/n_e$. It follows then from Equation 4.9 that

$$J_\pi = \sum_{\omega\in\mathcal{W}} \left(\frac{n_v(\omega)\,(1 - \mu^\omega(x_k))}{n_e} + \frac{1}{(1 - \mu^\omega(x_k))} \right) \tag{4.10}$$

We note that J_π is strictly convex and differentiable over $\mu^\omega \in [0, 1)$, and optimal values for the control inputs correspond with optimal solutions of the optimization problem

$$\min_{\mu\in[0,1)^{|\mathcal{W}|}} J_\pi$$

Such a solution is directly obtained from Equation 4.10 from the solution of the differential equation $\partial J_\pi / \partial \mu^\omega = 0$,

$$\mu^\omega(x_k) = \begin{cases} 1 - \left(\frac{n_e}{n_v(\omega)}\right)^{\frac{1}{2}}, & \text{if } n_e/n_v(\omega) < 1 \\ 0, & \text{otherwise} \end{cases} \tag{4.11}$$

Note that the *holding queue* follows the well-established practice of segmenting the search frontier into multiple queues [120]. We use the optimal policy we derived above to control the different queues based on their $w(s)$ values, ensuring each queue represents every category $\omega \in \mathcal{W}$ with positive probability. The queues are accessed sequentially, expanding all the nodes in the current queue before switching to the next. Also, the way we implement it, the queues are generated lazily, following a partial expansion of nodes whose successor falls in the subsequent queues.

4.5 Empirical Analysis

To evaluate the impact *Novelty approximation* and *open list control* have on width-based planners, we implemented different instantiations of BFWS(f_5): *complete* as described in Section 4.2, or *incomplete* if nodes with Novelty greater than a given bound are pruned [92]. We used the *Downward Lab* experiment module [134] on a server with Intel Xeon Processors (2 GHz) with a 1800 *sec* and 8 *GB time* and *memory* limit, respectively. All BFWS planners are implemented in C++ using the planning modules from *LAPKT* [117] and grounder from *Tarski* [52]. We use every benchmark in the IPC *satisficing* track to evaluate the correctness of the Novelty approximation $\hat{w}$ and the performance of new planners that use $\hat{w}$. If a domain has appeared over multiple IPCs, we used the problem set from the most recent IPC. We compare our new planners against notable *polynomial* planners: BFWS(f_5) with Novelty pruning and $\langle 1, 2\text{-C-M} \rangle$, a sequential polynomial planner [92], as well as two state-of-the-art planners DUAL-BFWS [93] and LAMA-*first* [120]. We show that the introduction of these methods has a significant impact on the performance of the BFWS algorithms.

4.5.1 Correctness of Novelty approximation.

We evaluate the reliability of the Novelty approximation by observing the effect on the rate of correct and incorrect (*lower* or *higher*) approximation of Novelty over varying sizes of sample $\bar{Z}$ and Bloom Filter r, scaled by a multiplicative factor δ. The Novelty approximation $\hat{w}$ is *correct* or *accurate* if $\hat{w}(s)$=$w(s)$. We limit the maximum size of the tuple evaluated to 3, as higher-order computations for exact Novelty w were infeasible within the practical constraints of time and memory. Thus, $w : S \rightarrow \mathcal{W}$, where $\mathcal{W}$=$[1, 4]$, and w=4 represents all nodes with w>3 . To distinguish the impact of sampling from that of Bloom Filter, we capture the results of Novelty approximation *with* and *without* Bloom Filter, hereafter, represented as $\hat{w}$ and $\hat{w}_{\bar{b}}$, respectively. We capture the statistics from 1200 solved instances in IPC satisficing benchmarks.

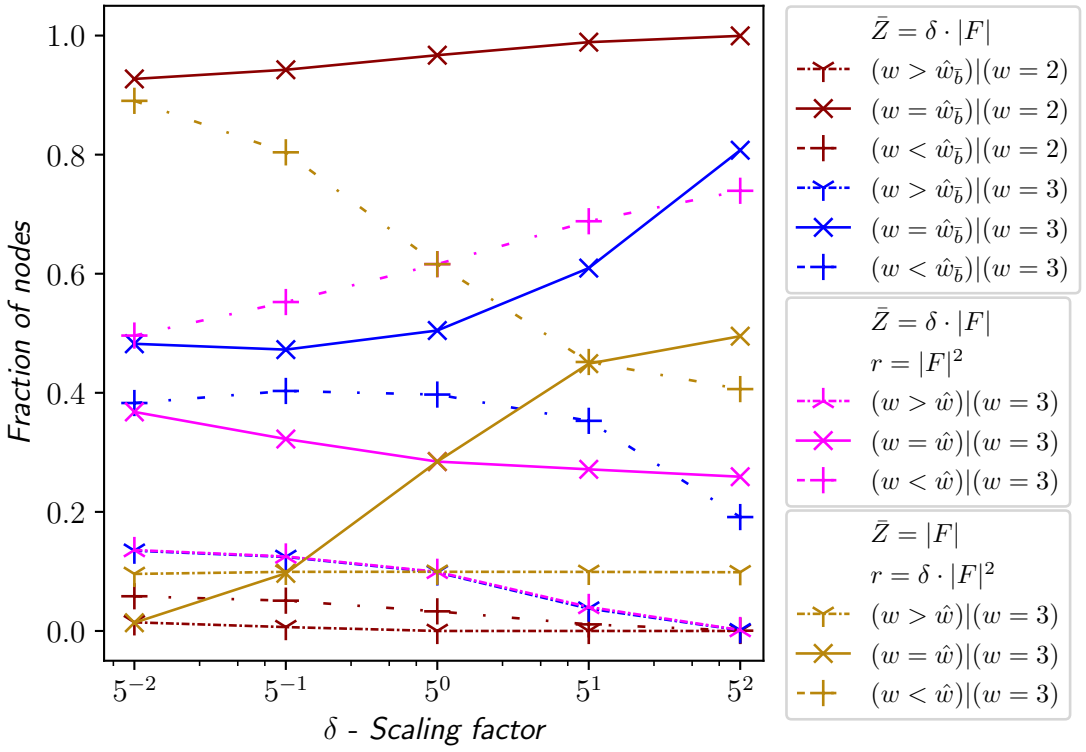


FIGURE 4.1: Variation in the rate of *accurate* ($w = \hat{w}$), *lower* ($w > \hat{w}$) and *higher* ($w < \hat{w}$) approximation of Novelty w over different sizes of sample($\bar{Z}$) and Bloom Filter(r).

From Fig. 4.1, we note that the rate of *correct* approximate Novelty ($w = \hat{w}_{\bar{b}}$) increases with sample size $\bar{Z}$, when Bloom Filters are not used. This backs up our analysis in Section 4.3 that the accuracy of Novelty approximation will likely increase with sample size. We also observe that the rate does not decrease below 1/2 even for $w = 3$, where

the sample is the order of $1/|F|^2$ smaller than the exhaustive set. This is a significant improvement over a trivial method of using a coin toss to determine whether or not a tuple t of size l is *new* in state s, which has probability 1/8 of selecting the *correct* novelty, given w=3 and guessing w>1, w>2, and w=3 sequentially with a coin toss.

While we note that the Novelty approximation without Bloom Filters performs satisfactorily in terms of correctness, it is still infeasible to store an exhaustive set of tuples $\beta_l(s)$ of size $O(|F|^l)$ when $l \geqslant 3$, for many IPC problems. As discussed in Section 4.3, we address this by using Bloom Filters to evaluate $\hat{w} \geqslant 3$. With this addition, we observe a slight decrease in the rate of *correct* Novelty approximation, which is the consequence of false positives, discussed in Section 4.2.2. Also, we observe that the trend along sample size is reversed, i.e., the rate of *correct* Novelty approximation now decreases with an increase in sample size. The trend aligns with the theoretical analysis in Section 4.3.2. On the other hand, increasing the size of Bloom Filter r improves the results as the false positive rate decreases.

4.5.2 Performance over benchmarks

Hereafter, we represent a particular configuration of BFWS planner as 'pI-(P|$\mathcal{L}$)$_{\bar{\omega}}$AC'. The *prefix* 'p' refers to the use of novelty-based pruning for nodes with $\hat{w}(s) > \bar{\omega}$. '$I$-' refers to BFWS called sequentially until the problem is solved over $\bar{\omega} \in [1, |F|]$. P$_{\bar{\omega}}$ refers to *BFWS(f_5)* planner with the set of possible Novelty categories $\mathcal{W} = [1, \bar{\omega} + 1]$. $\mathcal{L}_{\bar{\omega}}$ refers to BFWS(f_5) with the goal counting heuristics replaced by landmark counts [119], that is $f_{\mathcal{L}}$=$\langle w, h_L \rangle$. 'A' denotes that $\hat{w}(s)$ is used instead of $w(s)$, and 'C' denotes that BFWS is modified to control open list growth as described in Section 4.4. All 'AC' planners were run four times with different seeds, so we report the *mean and standard deviation* of statistics of interest.

We set the sample size $\bar{Z}$=$|F|$ so as to maintain a linear time complexity. We found that D_{max} values between 100 MB and 1GB had similarly good results for 'pI-P$_{\bar{\omega}}$AC', we show the results for $D_{max} = 500\ MB$. For the Bloom Filters size r, we didn't observe much variation between 100 KB / 8×10^5 $bits$ and 10 MB / 8×10^7 $bits$, with D_{max} = 1 GB. In our final implementation, we set an initial value of $r = |F|^2$, subject to increase when $\bar{\omega} r \times |V| < D_{max} \wedge |V| = |G| \times |F|$, and decrease when $\bar{\omega} r > D_{max} \wedge |V|$ = 1. A total of 103 instances out of 1691 used the *bank* of Bloom filters V, described in

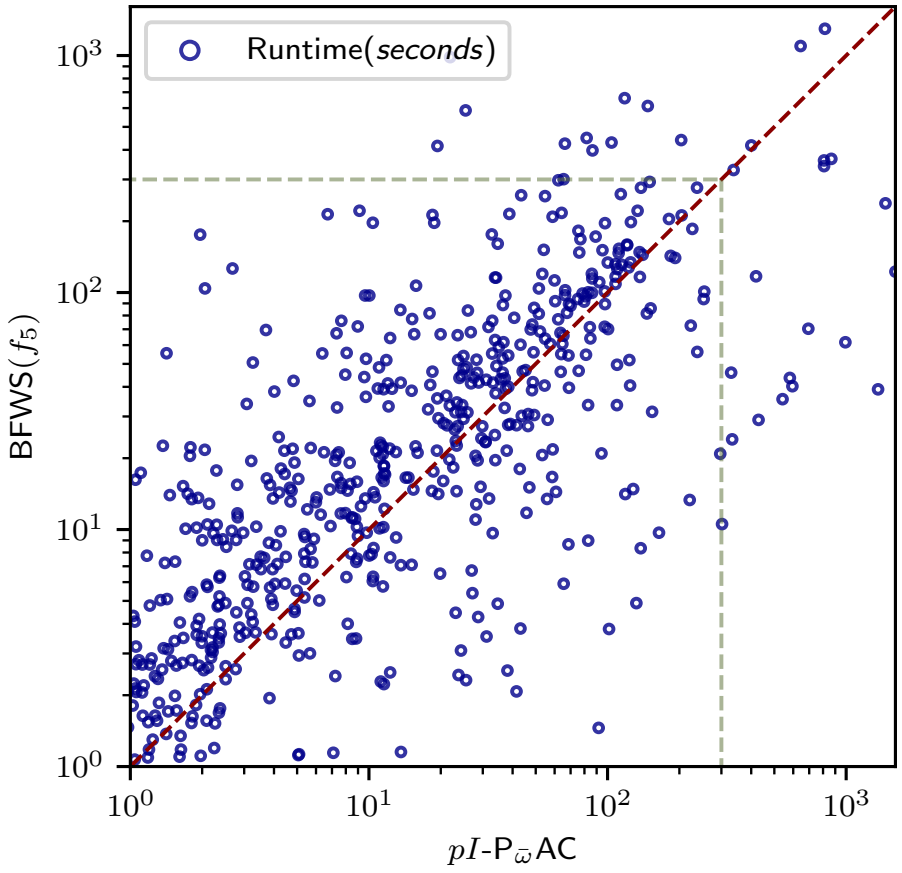


FIGURE 4.2: Pairwise comparison of runtime, over All IPC satisficing benchmarks, between BFWS(f_5) and pI-P$_{\bar{\omega}}$AC.

	$\bar{w} = 1$	$\bar{w} = 2$	$\bar{w} = 3$	$\bar{w} \geqslant 4$
# *Instances*	100.00 %	18.92%	3.68%	1.05%

TABLE 4.1: % of instances across all IPC satisficing benchmarks where a node of Novelty $\bar{w}$ was recorded in found plans.

Section 4.2, ensuring that Novelty computation does not exceed D_{max}. Lastly, we use the solution to the problem of minimizing P_f in Equation 4.1 to choose the number of hash functions as $K{=}\ln 2\ (r/q)$, where $q = \binom{|F|}{\omega}$.

Looking back at the motivation, a key driver for introducing the Novelty approximation was to enable Novelty computation for values greater than 2, which was infeasible for many IPC domains with the exact Novelty definition. The results for p-P3A in Table 4.2, show that our hypothesis was indeed correct as computing higher novelties with approximation improves coverage. This is substantiated in Table 4.1, which shows that $\approx 5\%$ of the solved instances had one or more nodes with $\bar{w}(s) \geqslant 3$ in the solution plan. Moreover,

the coverage of approximate planners with $\bar{\omega} = 2$, P_2A and p-P_2A, improves in comparison to P_2 and p-P_2, respectively, which indicates that there is no apparent demerit of using Novelty approximation. The improvement can be attributed to *polynomial* time and space complexity of $\hat{w}(s)$ allowing for additional search capacity.

Though BFWS($\hat{f}_5$) performs satisfactorily, it has a key shortcoming which impacts the search within the limited time environment, i.e., for large instances of domains with width $i > 2$, the BFWS search driven by the evaluation function $f_5 = \langle w, \#g \rangle$ exhausts all the available time in expanding nodes with $w(s) \leqslant 2$. Moreover, the issue gets compounded for domains with high branching factors as the open list doesn't fit within the memory bounds. We address both issues by applying the open list control discussed in Section 4.4. In our implementation, the *control* is not applied to child nodes with Novelty $w(s)$=1, as the maximum count of such nodes is small, $O(|F|)$, and have minimal impact on space. Note that this method will not cause the search to become incomplete. However, if we choose not to maintain the *holding queue*, we get a search that is *incomplete* and terminates early. Introducing the *open list control* in BFWS ($\hat{f}_5$) leads to a noticeable improvement in coverage of P_2AC and P_3AC which can be observed in Table 4.2. We do not report tables on plan length as it remains similar for all configurations.

At this point, we discuss a new planner, where we iteratively run the *polynomial* BFWS($\hat{f}_5$) with novelty-based pruning, sequentially increasing the number of Novelty categories $\mathcal{W}$ at each iteration, $\mathcal{W} = [1, \bar{\omega} + 1]$, over $\bar{\omega} \in [1, |F|]$. We denote the planner as 'pI-$P_{\bar{\omega}}AC$' where I stands for *iterative*. Informally, its major advantage is that it taps into the low polynomial space and time complexity of p-$P_{\bar{\omega}}AC$ with small $\bar{\omega}$ values as well as the greater coverage with larger $\bar{\omega}$. This can be observed in Table 4.2, which shows a significant jump in coverage compared to BFWS(f_5) with novelty-based pruning(p-P_2) and $\langle 1, 2\text{-C-M} \rangle$ (B3).

The coverage is also higher than the state-of-the-art LAMA-*first* (B1) and DUAL-BFWS (B2). Moreover, from Fig. 4.2, we note that the 'pI-$P_{\bar{\omega}}AC$' planner has better runtime performance than BFWS(f_5), the winner of the Agile track (IPC 2018). It solved 59 more instances than BFWS(f_5) across every IPC satisficing benchmarks with a 300 *sec time* and 8 *GB memory* limit. At the same time. Fig. 4.3 confirms that the space and time consumption is much less than the baseline BFWS planners. It is worth pointing that 'pI-$P_{\bar{\omega}}AC$' is probabilistically *incomplete*. Also, we did not observe any difference

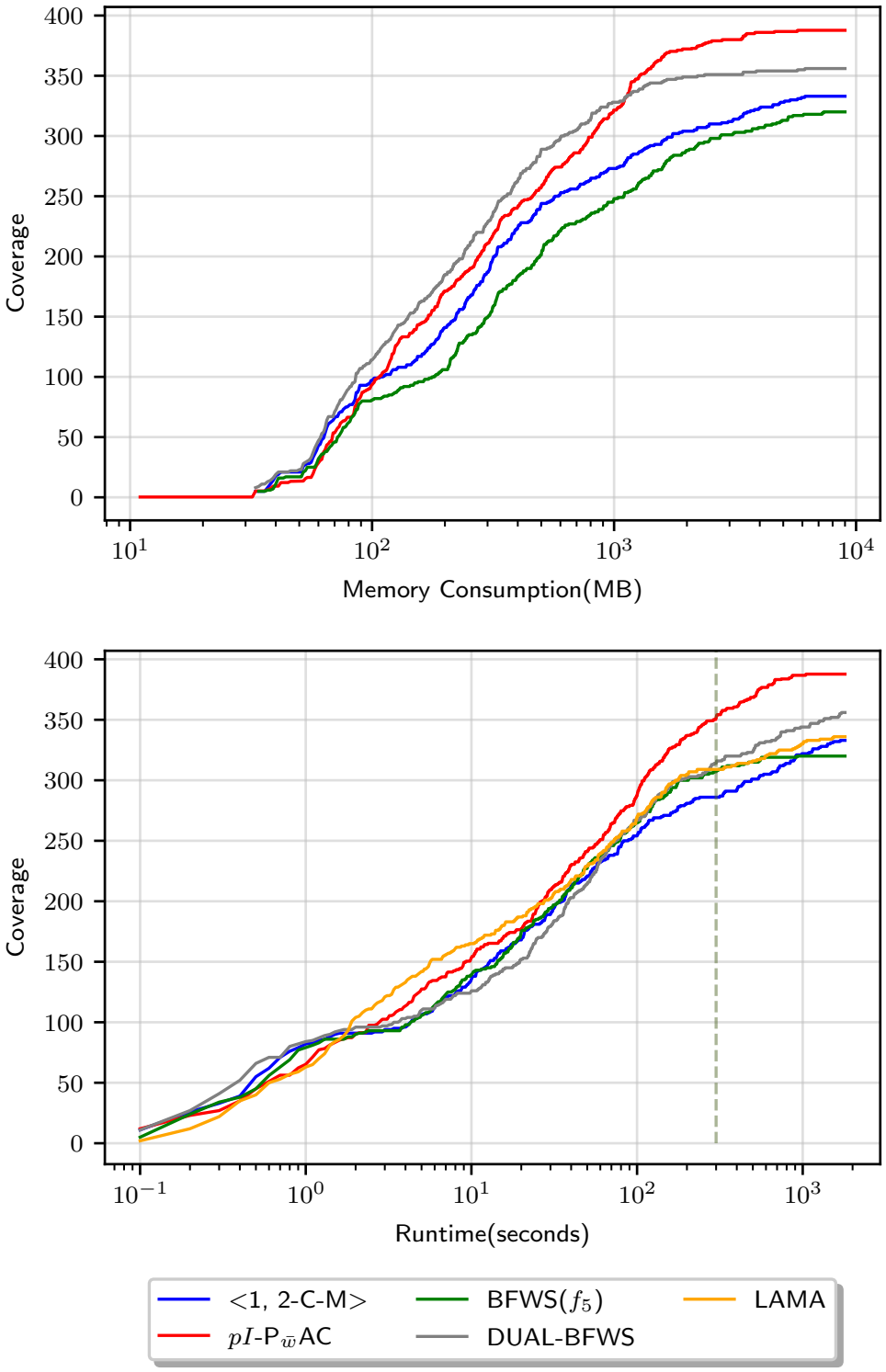


FIGURE 4.3: *Coverage* over *memory(MB)* and *time(seconds)* on IPC 2014 and 2018 satisficing benchmarks. The dotted vertical line represents 300 seconds.

in coverage of '*pI*-P$_{\bar{\omega}}$AC' *with* or *without* the *holding queue*, as the nodes pruned at one iteration get selected in subsequent iterations with positive probability.

domain	B1	B2	P_2	P_2A	P_2AC	P_3A	P_3AC
agricola (20)	12	8	11	11±1.0	12±1.7	15±0.8	**16±1.3**
airport (50)	34	**47**	46	46±0.6	46±0.6	44±0.0	44±0.6
assembly (30)	**30**	**30**	**30**	**30±0.6**	29±1.0	**30±0.6**	**30±0.6**
caldera (20)	16	**20**	15	16±1.0	**20±0.5**	16±1.0	18±0.5
cavediving (20)	7	7	7	7±0.0	**8±0.6**	**8±0.0**	**8±0.5**
childsnack (20)	6	**10**	0	4±1.3	5±1.7	3±1.9	6±0.6
citycar (20)	5	**20**	5	5±0.0	**20±0.0**	5±0.0	**20±0.6**
data-network (20)	13	11	9	12±1.7	**19±1.0**	11±2.5	18±0.5
depot (22)	20	**22**	**22**	**22±0.0**	**22±0.0**	**22±0.0**	**22±0.0**
flashfill (20)	14	**16**	12	14±2.4	14±0.6	14±2.2	14±1.0
floortile (20)	**2**	**2**	1	**2±0. 5**	**2±0.0**	**2±0.0**	**2±0.0**
hiking (20)	**20**	12	12	14±2.1	8±0.8	18±1.0	**20±0.5**
maintenance (20)	11	**17**	**17**	16±0.5	16±0.6	16±0.5	**17±0.5**
mprime (35)	**35**	**35**	32	30±0.6	**35±0.0**	31±0.5	34±0.8
mystery (30)	**19**	**19**	**19**	**19±0.0**	**19±0.0**	**19±0.5**	**19±0.5**
nomystery (20)	11	**19**	13	14±1.0	12±1.0	13±0.5	14±1.0
nurikabe (20)	9	14	**16**	14±0.6	15±1.3	14±0.6	15±2.1
org-synth-split (20)	**12**	11	5	6±0.5	3±0.8	7±1.0	5±1.4
parcprinter (20)	**20**	16	9	5±1.0	5±1.9	5±0.8	6±1.0
pathways-neg (30)	24	**30**	23	**30±0.6**	29±1.5	**30±0.6**	29±0.5
pegsol (20)	**20**	**20**	**20**	**20±0.0**	**20±0.5**	**20±0.0**	**20±0.0**
pipesworld-nt (50)	43	**50**	**50**	**50±0.0**	**50±0.0**	**50±0.0**	**50±0.0**
pipesworld-t (50)	**43**	38	**43**	42±0.5	42±0.6	42±0.5	**43±0.8**
psr-small (50)	**50**	**50**	48	49±0.5	49±0.5	**50±0.0**	**50±0.0**
rovers (40)	**40**	37	39	**40±0.0**	**40±0.0**	**40±0.0**	**40±0.0**
satellite (36)	**36**	31	27	30±0.8	32±0.6	30±0.5	30±0.0
schedule (150)	**150**	149	149	149±1.0	149±0.8	149±1.0	**150±0.6**
settlers (20)	**18**	8	7	6±1.0	12±0.6	6±1.3	10±0.0
snake (20)	5	12	**19**	16±0.5	15±0.8	17±0.5	17±0.5
sokoban (20)	**19**	17	14	15±0.5	10±1.0	16±0.5	14±0.8
spider (20)	**16**	14	13	15±1.0	14±1.0	15±1.0	14±1.3
storage (30)	20	28	29	**30±0.6**	**30±0.6**	**30±0.6**	**30±0.5**
termes (20)	**16**	9	9	10±0.0	8±0.6	9±1.0	8±1.3
tetris (20)	16	16	**20**	**20±0.0**	**20±0.0**	**20±0.0**	**20±0.0**
thoughtful (20)	15	**20**	**20**	**20±0.0**	**20±0.0**	**20±0.0**	**20±0.0**
tidybot (20)	17	18	19	**20±0.0**	**20±0.5**	**20±0.0**	**20±0.0**
tpp (30)	30	29	29	**30±0.6**	**30±0.0**	29±0.0	**30±0.0**
transport (20)	16	**20**	**20**	**20±0.0**	**20±0.0**	**20±0.0**	**20±0.0**
trucks-strips (30)	**18**	16	9	9±0.8	9±1.3	9±0.8	10±1.4
Total (1691)	1456	1496	1436	1455±8.7	1476±4.2	1463±8.9	1502±4.9

TABLE 4.2: Coverage over all satisficing benchmarks from IPCs: *complete* — B1: LAMA-*first*, B2: DUAL-BFWS, 'P...'. $P_{\bar{\omega}}$ refers to BFWS(f_5) planner with $\mathcal{W} = [1, \bar{\omega} + 1]$, $\mathcal{L}_{\bar{\omega}}$ is BFWS($f_{\mathcal{L}}$) , which uses *Landmarks* , '*I*-' stands for *Iterative*, 'A' for *approximate*, and 'C' for *control* over *open list*. The *mean coverage* is shown along with the *standard deviation* for the planners using random sampling over four different seeds. Domains which all planners fully solve are omitted. The best results are highlighted in bold and red.

Discussion. We show that *approximate Novelty search* greatly improves the performance over baseline BFWS planners. The ability to compute $\hat{w} > 2$ using Novelty approximation, within practical time and memory constraints, allows us to use the '*pI*-$P_{\bar{\omega}}$AC' configuration that beats the state-of-the-art. This is impressive for a sequential polynomial planner, which uses simple goal counting heuristics $\#g(s)$ and relaxed plan counter $\#r(s)$ along with $\hat{w}$ to direct the search. Also, we can observe that certain domains were affected more than others. Specifically, the domains *citycar, data-network, hiking, and satellite* benefited significantly.

domain	p-P$_2$	B3	p-P$_2$A	p-P$_3$A	pI-P$_{\bar{\omega}}$AC	pI-$\mathcal{L}_{\bar{\omega}}$AC
agricola (20)	10	15	10±0.6	15±0.8	**16±1.3**	12±0.5
airport (50)	46	**47**	46±0.6	45±1.0	46±0.6	46±0.5
assembly (30)	**30**	**30**	**30±0.5**	**30±0.0**	**30±0.0**	**30±0.0**
caldera (20)	19	**20**	**20±0.5**	18±0.5	**20±0.5**	**20±0.0**
cavediving (20)	1	8	2±2.9	8±0.5	**9±1.0**	8±0.5
childsnack (20)	0	2	5±0.5	6±0.6	8±1.3	8±1.3
citycar (20)	**20**	**20**	**20±0.5**	5±0.0	**20±0.0**	**20±0.0**
data-network (20)	16	14	17±1.0	16±0.6	18±0.6	18±0.6
depot (22)	**22**	**22**	**22±0.0**	**22±0.0**	**22±0.0**	**22±0.0**
flashfill (20)	15	9	14±2.4	14±1.9	14±1.0	14±1.0
floortile (20)	0	1	1±0.0	**2±0.5**	**2±0.0**	**2±0.0**
hiking (20)	9	13	12±1.8	**20±0.0**	**20±0.0**	19±0.5
maintenance (20)	**17**	**17**	16±0.5	16±0.5	**17±0.5**	**17±0.5**
mprime (35)	**35**	**35**	**35±0.0**	32±0.8	**35±0.0**	**35±0.0**
mystery (30)	**19**	18	**19±0.0**	**19±0.5**	**19±0.5**	**19±0.5**
nomystery (20)	13	13	12±1.7	14±0.6	15±1.0	18±1.4
nurikabe (20)	**16**	**16**	14±0.5	14±0.6	15±1.0	14±0.6
org-synth-split (20)	4	3	4±0.5	6±1.0	7±0.0	6±0.5
parcprinter (20)	9	16	6±1.0	5±1.0	8±0.0	6±0.5
pathways-neg (30)	24	27	**30±0.5**	**30±0.6**	**30±0.0**	**30±0.0**
pegsol (20)	5	**20**	12±1.5	18±0.5	**20±0.0**	**20±0.0**
pipesworld-nt (50)	**50**	**50**	**50±0.0**	**50±0.0**	**50±0.0**	**50±0.0**
pipesworld-t (50)	41	39	41±1.5	42±0.5	42±1.2	**43±1.5**
psr-small (50)	31	46	34±1.3	43±0.8	49±0.5	48±0.6
rovers (40)	39	38	**40±0.0**	**40±0.0**	**40±0.0**	**40±0.0**
satellite (36)	27	31	32±0.5	30±0.5	34±0.6	34±0.6
schedule (150)	149	149	149±1.0	149±1.0	149±1.0	149±1.0
settlers (20)	10	11	9±1.0	6±1.0	12±0.6	17±1.3
snake (20)	18	3	16±0.5	17±0.5	**20±0.5**	**20±0.5**
sokoban (20)	13	11	15±1.0	15±1.0	14±1.0	15±0.5
storage (30)	**30**	29	**30±0.6**	**30±0.6**	**30±0.5**	**30±0.6**
termes (20)	1	6	2±0.5	6±1.9	7±1.4	10±1.3
tetris (20)	**20**	18	**20±0.0**	**20±0.0**	**20±0.0**	**20±0.0**
thoughtful (20)	**20**	**20**	**20±0.0**	**20±0.0**	**20±0.0**	**20±0.0**
tidybot (20)	**20**	**20**	**20±0.0**	**20±0.0**	**20±0.0**	19±0.5
tpp (30)	**30**	**30**	**30±0.0**	**30±0.0**	**30±0.0**	**30±0.0**
transport (20)	**20**	**20**	**20±0.0**	**20±0.0**	**20±0.0**	**20±0.0**
trucks-strips (30)	11	8	12±1.8	11±1.3	12±1.3	12±0.8
Total (1691)	1414	1456	1438±5.9	1462±8.0	**1524±2.5**	1516±5.0

TABLE 4.3: Coverage over all satisficing benchmarks from IPCs: *polynomial incomplete* — B3: $\langle 1, 2\text{-C-M}\rangle$ and 'p-P...'. P$_{\bar{\omega}}$ refers to BFWS(f_5) planner with $\mathcal{W} = [1, \bar{\omega}+1]$, $\mathcal{L}_{\bar{\omega}}$ is BFWS($f_{\mathcal{L}}$) , which uses *Landmarks* , 'I-' stands for *Iterative*, 'A' for *approximate*, and 'C' for *control* over *open list*. The *mean coverage* is shown along with the *standard deviation* over four different seeds for the planners using random sampling. Domains that all planners fully solve are omitted. The best results are highlighted in bold and red.

We found that the *open list control* significantly benefited the domains *citycar and data-network*, which have a high branching factor but are solvable with $\hat{\omega} \leqslant 2$. *Citycar* in particular was fully solvable with $\hat{\omega} = 1$ and discarding nodes with $w(s) > 1$ didn't impact the order of expansion. *Hiking* and *satellite* on the other hand required expansion of nodes of $w(s) > 2$, and the increased coverage highlights the importance of policy-based control of different Novelty categories in the open list. *Childsnack and Floortile* showed no improvement, which is a combined effect of high width and the fact that our goal count heuristic $\#g(s)$ is not informed enough.

4.6 Approximate Novelty Search in *International Planning Competition 2023*

The International Planning Competition of 2023 included many entirely new domains not seen in the previous IPCs, including Labyrinth — a game where the agent must escape from a maze, Quantum Circuit — which requires the solver to map logical quantum circuits to physical qubits, Recharging Robots — a coordination problem that requires observation robots to schedule their recharging times such that the security levels are maintained, and lastly, the classic Rubik's Cube. The varying characteristics of the domains, hard-to-ground, impractical Novelty value of states in feasible plans, and the complex structures of dependencies between the fluents of states in feasible plans presented a challenge to our planners.

We registered two planners in the competition, both implementing Approximate Novelty Search. One uses the grounder available in the Tarski library, which uses an off-the-shelf solver (Gringo), and the other the Fast Downward Grounder. We did this to compare the grounding performance of the two approaches in a blind test for the methods and highlight the *potentially significant* impact of efficient grounding methods in the ever-so-more competitive planning competitions. The size of the ground representation directly impacts the efficiency of the planners due to the overheads from the management and processing of a large number of ground actions and fluents. Hence, state-of-the-art planners that work on *ground* representations use some method that tries to reduce the size of the grounding. Both configurations were submitted to the *agile* track of the competition, as the tighter time constraints result in grounding time being more valuable. Furthermore, we also entered the *satisficing* track, but only with the latter configuration, to avoid redundant experiments.

Although we did not win the competition, we are happy to report that our planners performed very well on many of the domains included in the competition. In this section, we explain the configuration of the Approximate Novelty Search planner submitted in the IPC, summarize the results, and justify the observed performance in each domain.

Sequential *polynomial approximate* BFWS(f_5). In this planner, we make sequential calls to the *polynomial approximate* BFWS(f_5) with novelty-based pruning until we run out of time. Each *polynomial approximate* BFWS(f_5) is a p-P$_{\bar{\omega}}$AC planner, where

	ApxNovelty (*agl*)	ApxNovelty (*sat*)
Folding	4	5
Labyrinth	11(0)	15(0)
Quantum Layout	20	20
Recharging Robots	10(6)	11(8)
Ricochet Robots	14	18
Rubik's Cube	4	5
Slitherlink	3	4

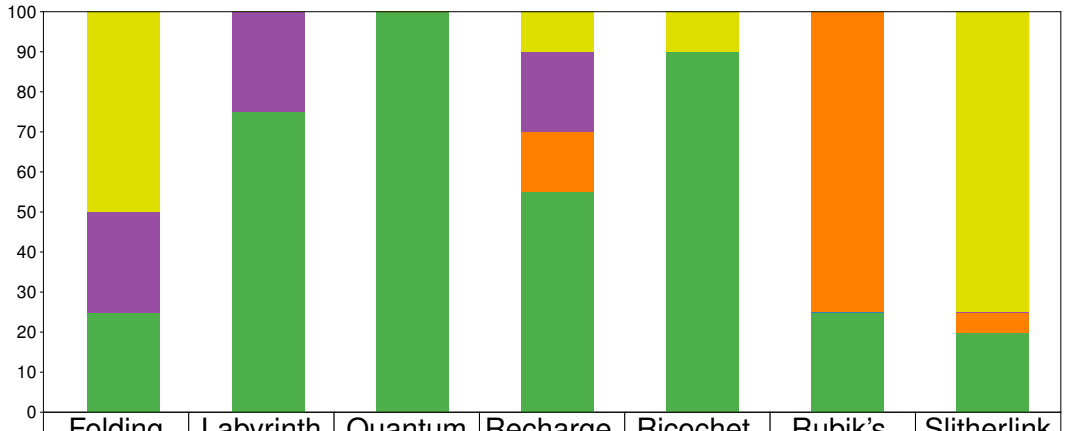
TABLE 4.4: Coverage of Sequential *polynomial approximate* BFWS(f_5) (ApxNovelty) in Agile and Satisficing tracks. The numbers in brackets represent the coverage in the IPC 2023 for domains that encountered preprocessing error[3].

the set of Novelty categories considered in the computation of Novelty is $\mathcal{W} = [1, \bar{\omega} + 1]$ and nodes with $\hat{w}(n) > \bar{\omega}$ are pruned.

Performance in IPC 2023 benchmarks. Table 4.4[3] shows the coverage of Sequential polynomial approximate BFWS on the IPC 2023 instances, revealing that increasing the time limit from 300 seconds in the *Agile* track to 30 minutes in the *satisficing* track only slightly improves the coverage in 5 of the 7 domains. To further examine the *satisficing* results, we use two bar plots to illustrate the characteristics of the planning runs. Figure 4.4 presents the performance profile of the planner in terms of the percentage of the problems solved in each domain and a breakdown of the reason for failure in unsolved instances — load memouts(lm%), load timeouts(lt%), search memouts(sm%), and search timeouts(st%). The figure helps us identify which component of the planner's algorithm stack is challenged by a specific domain. Figure 4.5 shows the distribution of the minimum Novelty bound of the polynomial planner at which it finds a solution to the instance. We now explain the performance of the planner on individual domains.

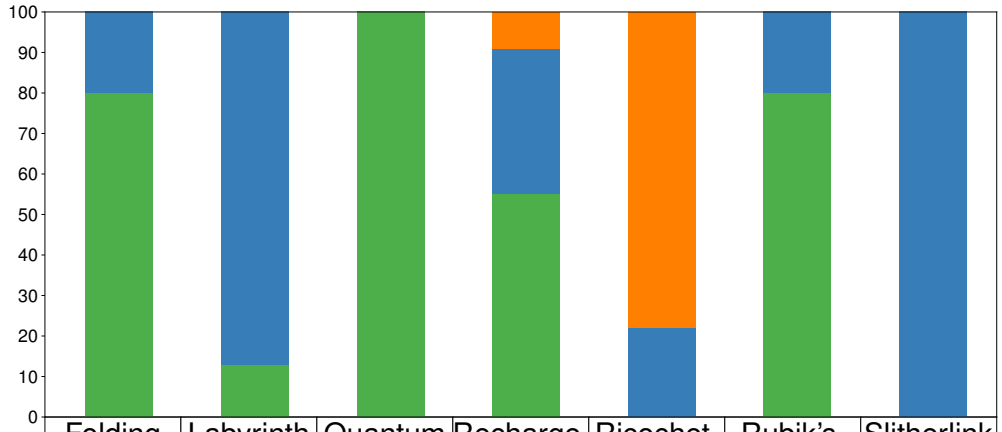
Quantum Layout domain received the Outstanding Domain Submission Award out of all the submitted domains in the IPC 2023. It is an exciting application of planning on a problem of practical significance. Our planner performs best in this domain among all the participants of IPC 2023 in the *Agile track*. A detailed analysis of the planner's performance reveals this is not a coincidence. Our polynomial planner solved 95% instances with a Novelty bound of 1. The sole outlier only required one more iteration of the

[3]Our integration with FAST DOWNWARD Grounder failed in Labyrinth and Recharging-robots instances with *single-goal atoms*, resulting in the planner terminating unexpectedly at the step when the search engine's data structures are initialized. The error prevented us from accurately analyzing and justifying the performance of Approximate Novelty Search in the Satisficing Track using the IPC 2023 results. Hence, we redid the experiments on the two domains. We executed the experiments on a server using Intel Xeon Processors (2 GHz) with 1800 sec and 8 GB time and memory limit.



	Folding	Labyrinth	Quantum	Recharge.	Ricochet.	Rubik's.	Slitherlink
sm%	50	0	0	10	10	0	75
lm%	25	25	0	20	0	0	0
st%	0	0	0	15	0	75	5
lt%	0	0	0	0	0	0	0
s%	25	75	100	55	90	25	20

FIGURE 4.4: Plot showing the *performance profile* of the Sequential *polynomial approximate* BFWS. s% represents the percentage of solved instances, and the exit codes of unsolved instances are captured as — load memouts(lm%), load timeouts(lt%), search memouts(sm%), where "load" refer to the preprocessing phase of *parsing and grounding*.



	Folding	Labyrinth	Quantum	Recharge.	Ricochet.	Rubik's.	Slitherlink
$(\overline{\omega} \geq 3)$%	0	0	0	9	78	0	0
$(\overline{\omega} = 2)$%	20	87	0	36	22	20	100
$(\overline{\omega} = 1)$%	80	13	100	55	0	80	0

FIGURE 4.5: Plot showing the distribution of *minimum Novelty bound* in Sequential *polynomial approximate* BFWS, *necessary* to find a feasible plan in the *solved* instances.

polynomial planner with a bound of 2. Such a consistent finding across all 20 instances suggests that the structure of the instances — reachability relation between fluents in plans — exhibits characteristics that align with the concept of problem width [91] in width-based planning algorithms. Since this is a problem of practical interest, we believe that further study to explore the possibility of a low upper bound on the value of the

Novelty measure is warranted.

Ricochet-robots is another domain where the polynomial Novelty planner performs well. However, in contrast to Quantum Layout, this problem domain requires expanding nodes of Novelty greater than 2. As noted earlier, the computation of Novelty > 2 in the IPC instances is generally impractical, except for tiny instances. However, the fact that we solve 95% of the instances in this domain demonstrates the usefulness of approximation methods that trade-off accuracy for computational guarantees. Here, the *Novelty approximation* enables the planner to compute Novelty measure *approximately* but in *linear* time and consuming a fixed amount of memory when it is practically infeasible to do so precisely. The observed lower bound on the highest Novelty of a state in any plan is 3. This finding is noteworthy as Ricochet-robots is the only domain from the IPCs where most instances require Novelty computation that is impractical to do *exactly*. At the same time, the lower bound on the highest Novelty of states in any plan is reachable with Sequential *polynomial approximate* BFWS.

Labyrinth is a domain with big instances, also considered to be *hard-to-ground*, in which most problems have more than half a million grounded actions. Hence, it is unsurprising that many instances failed in the grounding phase. The polynomial Novelty planner solved the remaining instance, which could be grounded, with the Novelty bound of 2.

In the instances of **recharging robots**, our planner solved 50% of the instances. All except one instance were solved within the Novelty bound of 2 and the outlier with a Novelty bound of 3. Most instances that could not be solved ran out of memory, likely because of the large number of ground actions, which sometimes exceeded a million.

In **Folding, Rubik's Cube, and Slitherlink**, many instances ran out of computational resources of time and memory while searching for a solution at or below the Novelty bound of 3. This observation points to the possibility that the region of state space that is reachable at low Novelty bounds of 1, 2, and 3 is very large and stresses the open and closed lists — data structures that store a representation of the explicit search tree and the explored state space for us.

Overall, the results show that our planners performed excellently in the IPC 2023 instances, particularly in the Agile Track. Despite the bug that eliminated the coverage on Labyrinth and significantly reduced the coverage on Recharging Robots, the planner still managed to rank seventh out of twenty-three planners. Looking at the new results,

we believe that the planner would have ranked among the top two in the Agile Track if not for the unfortunate bug that affected the FD Grounding and LAPKT integration.

4.7 Grounding Schematic Representation with Gringo

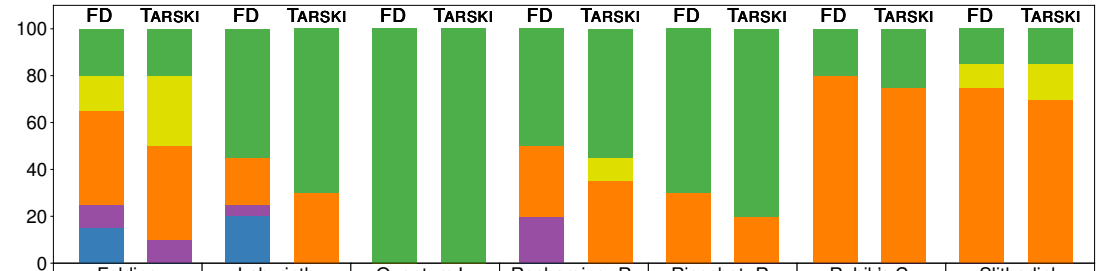
Lifted representations of planning problems, like PDDL [60] allows using first-order logic variables to represent actions and fluents compactly. However, most planners, including width-based planners, work only on *grounded* representations that require substituting the first-order logic variables by type consistent constants. A *trivial* substitution mechanism would generate a ground representation that is exponential on the arity of actions and predicates [68]. The size of the ground representation directly impacts the efficiency of the planners due to the overheads from the management and processing of a large number of ground actions and fluents. Hence, state-of-the-art planners that work on *ground* representations use some method that tries to reduce the size of the grounding. The grounder available in the Tarski library [52] generates a logic program [17] and uses it to over-approximate the set of reachable actions based on the compilations defined by Helmert [67]. However, in contrast to the approach taken by the FastDownward grounder, Tarski uses an off-the-shelf solver (Gringo) [55] that grounds a logic program whose answer set captures all applicable actions in the delete-free relaxation [112, 67]. We believe that the generally more efficient grounding of action schemas based on Gringo would give *grounded* planners an edge. To confirm this hypothesis, we integrated our planner with both Fast Downward and Tarski grounders and submitted them both in the *agile* track. In this section, we analyze and compare the performance of the two planners in the competition, mainly focusing on the contribution of the more efficient grounding of Tarski[4].

Performance in IPC 2023 benchmarks. Table 4.5 helps us understand the performance based on which particular constraint, either time or memory limits, stresses out one grounder more quickly than the other. The Fast Downward grounder exceeded time and memory limits with equal frequency and could not ground 14 instances. Gringo, on the other hand, grounded more instances than Fast Downward giving

[4]The Tarski grounder encountered failure on many instances of Folding, Labyrinth, and Recharging Robots in the IPC 2023 runs. We could not reproduce the error locally, which prevented us from comparing the performance of the Fast Downward and Tarski(Gringo) grounders. Hence, we redid the experiments on the three domains for both FD and Tarski Grounders. We executed the experiments on a server using Intel Xeon Processors (2 GHz) with 1800 sec and 8 GB time and memory limit.

TARSKI \ FD	Success	Timeout	Memout
	126	7	7
Success 138	126	6	6
Timeout 0	0	0	0
Memout 2	0	1	1

TABLE 4.5: A pairwise comparison of TARSKI and FAST DOWNWARD(FD) grounders, showing the *count of instances* with the same(diagonal elements) or different(non-diagonal element) exit status — *grounding success* and *failures*, including *memouts* and *timeouts*. The individual profiles of TARSKI and FD are shown in different colors.



	Folding		Labyrinth		Quantum L.		Recharging. R.		Ricochet. R.		Rubik's C.		Slitherlink	
s%	20	20	55	70	100	100	50	55	70	80	20	25	15	15
st%	15	30	0	0	0	0	0	10	0	0	0	0	10	15
sm%	40	40	20	30	0	0	30	35	30	20	80	75	75	70
lt%	10	10	5	0	0	0	20	0	0	0	0	0	0	0
lm%	15	0	20	0	0	0	0	0	0	0	0	0	0	0

FIGURE 4.6: Plot showing the *performance profile* of the two Approximate Novelty Search planners using FAST DOWNWARD(FD) and TARSKI(GRINGO) grounders. s% represents the percentage of solved instances, and the exit codes of unsolved instances are captured as — load memouts(lm%), load timeouts(lt%), search memouts(sm%), search timeouts (st%), where "load" refer to the preprocessing phase of *parsing and grounding*.

the planner an edge on at least 12 problems. While grounding more instances does not equate to solving more, it certainly improves the odds. In addition to being able to ground more instances, Figure 4.7 shows a clear runtime advantage of using GRINGO. Here, we observe that it could ground many problems with less than four times runtime in a clear win for the TARSKI approach to grounding. This observation solidifies our belief that *grounding using GRINGO* is *generally more efficient*.

We conclude by looking at Figure 4.6, which presents the overall performance profiles, including search performance, of the two Approximate Novelty Search planners in the Agile Track. We observe that the efficient grounding of TARSKI allows us to solve three more instances in Labyrinth, which is a *hard-to-ground* instance, and a total of seven additional problems overall. We believe that the 10% improvement in coverage, from 66 to 73, by using a more efficient grounding method based on GRINGO, is significant.

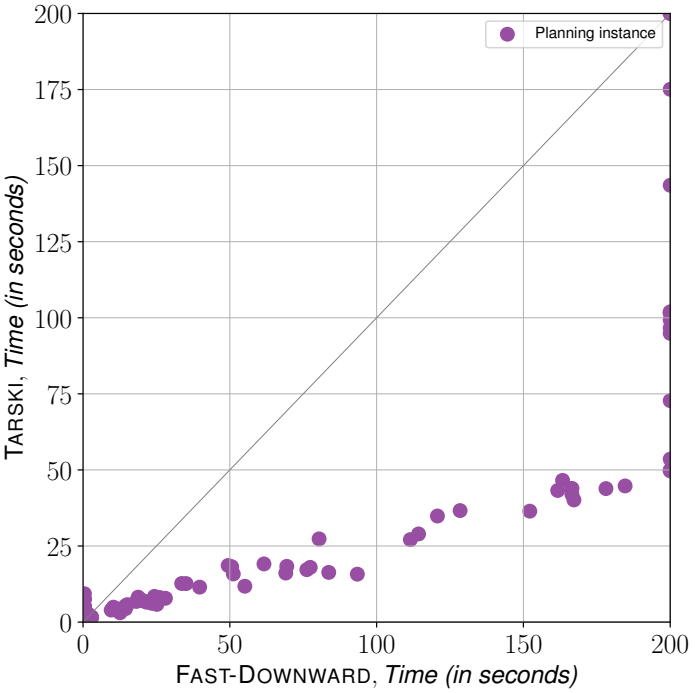


FIGURE 4.7: A pairwise comparison of the parsing and grounding times of TARSKI(GRINGO) and FAST DOWNWARD(FD) grounders.

The results confirm the superiority of using off-the-shelf ASP solvers for grounding the logic program formulated by Helmert [67]. These results also show that grounding is a critical bottleneck for classical planning methods. Managing the intractability, in general, of grounding with scalable algorithms provides a significant edge in the ever-so-more competitive planning competitions. The development of new methods for grounding that scale up and are better integrated with SAT and heuristic search methods for planning is a problem that has long been neglected until very recently [29, 76] and requires attention.

4.8 Conclusion

The proposed methods of Novelty approximation and open list control in *BFWS* not only have a positive impact on coverage but also on the overall time and space complexity of the search, resulting in new state-of-the-art planners over satisficing benchmarks from every IPC. These results strongly suggest that probabilistically complete search algorithms are a promising research direction in classical planning. This is especially crucial

in limited time and memory environments where the search must work within hard constraints on time and memory. However, we must note that approximate Novelty search is no silver bullet, and specific domains, including *Childsnack and Floortile*, remain unsolved. We hope this work brings about the insights to develop the next generation of classical planners that scale up better as the intractability of the benchmarks ramps up and tackles the inherent limitations of BFS.

Part III

Lazy Clause and Constraint Generation

"Honest disagreement is often a good sign of progress."

M.K. Gandhi

5

Lifted Sequential Planning with Lazy Constraint Generation

In this part of the thesis, we study the possibilities that arise from the use of Lazy Clause Generation (LCG) based approaches to Constraint Programming (CP) for tackling *sequential classical planning*. We propose a novel CP model based on seminal ideas on so-called lifted causal encodings for planning as satisfiability, which does not require grounding, as choosing groundings for functions and action schemas becomes an integral part of the problem of designing valid plans. This encoding does not require encoding frame axioms and does not explicitly represent states as decision variables for every plan step. We also present a propagator procedure that illustrates the possibilities of LCG to widen the kind of inference methods considered to be feasible in planning as (iterated) CSP solving. We test encodings and propagators over classic IPC and recently proposed benchmarks for lifted planning, and report that for planning problem instances requiring fewer plan steps, our methods compare very well with the state-of-the-art in optimal sequential planning.

5.1 Introduction

Like R. Frost's Traveller in the Woods, this work finds its way back to a crossroads in the development and study of approaches to planning as satisfiability. One way is of so-called *GraphPlan* encodings (linear and parallel), the other being that of so-called *causal* encodings (ground and lifted) invented by Kautz, McAllester, and Selman (KMS) [83]. Unlike Frost's poem, though, lifted causal encodings are an approach seldom followed in the literature in automated planning. Inspired by the formulation of *nonlinear* or partially-ordered planning of McAllester and Rosenblitt [98], these encodings have polynomial size over the lower bound on the number of plan steps in feasible plans. Importantly, they do away entirely with the need for explanatory frame axioms. Yet these very desirable properties follow from the premise of not having to ground actions and predicates first, as otherwise, the unavoidable exponential blowouts obliterate any practical differences with GraphPlan encodings.

In this research, we realize this potential by tapping into the power of the Lazy Clause Generation (LCG) [108], a ground-breaking technology that unifies propositional satisfiability (SAT) and Constraint Programming (CP) and allows representing implicitly large tracts of complex systems of constraints by suitably defined inference procedures or *propagators*. These lazily generate new constraints to record violations by assignments to decision variables and propagate information following the consistency of assignments and constraints to tighten the domains of variables. This, in addition to very sophisticated and performant modeling tools and solvers [113], provides us with the foundations to develop scalable planners that follow the path laid by KMS lifted causal encodings. We have found these planners to outperform state-of-the-art optimal planning algorithms on benchmarks designed to be *hard to ground* [29], while standing their ground on the IPC benchmarks.

Chapter Outline. We begin by explaining a precise formulation of the planning problems of interest to us and their solutions. We then introduce a formalism to describe the structure of states and actions that is based on Functional STRIPS [51]. We assume all atoms in precondition and effects are equality atoms over suitably defined function symbols and constant terms. These ground *domain theories* are then lifted [98]. With these preliminaries in place, we introduce our encoding, proving its validity and characterizing its complexity. After that, we explain how we leverage state-of-the-art CP

solvers to implement our encoding efficiently. We then present a method to do a *concise* transformation of planning instances represented in PDDL to FSTRIPS. We end with an evaluation of several planners built on top of our encoding and propagators, along with a brief analysis of related work and a discussion of the significance and potential of this research.

5.2 Formulation of Planning Problems

A problem planning instance (PPI) is given by a tuple $P = (S, A, \rightarrow, s_0, S_G)$ where S and A are finite sets of states and actions, $\rightarrow \subset S \times A \times S$ is the *transition relation* which is *right-unique*, where $s \rightarrow_a s'$ indicates that s' is reachable from s via a, $s_0 \in S$ is the *initial state* and $S_G \subset S$ is the set of designated *goal states*. We say that a PPI *admits* a trajectory $\sigma = s_0, a_1, s_1, \ldots, s_{i-1}, a_i, s_i$ iff $s_{i-1} \rightarrow_{a_i} s_i$ for every $i > 0$. In this work, the notion of *planning problems* is that of *optimization problems* where we seek *sequences* $\sigma = s_0, a_1, s_1, \ldots, s_{k-1}, a_k, s_k$, that minimize length$(\sigma) := k$, and satisfies $s_k \in S_G$. The set of σ sequences that satisfy $s_k \in S_G$ is referred to as the set of *valid plans* Π_P. The optimal cost of P is thus $c^* := \min_{\sigma \in \Pi_P} \text{length}(\sigma)$. When $\Pi_P = \varnothing$, we say that P is infeasible and optimal cost is $c^* = \infty$. We observe that $|S| - 1$ is a trivial upper bound on c^* when $\Pi_P \neq \varnothing$. Non-trivial, feasibly computable upper bounds are known [1] but only for PPIs with special structure.

5.2.1 Factored Planning Problems

A long-recognised and adopted strategy to deal with large $|S|$ is that of *factoring* states and actions in PPIs as a preliminary step to develop algorithms that can deal with large scale problems [110, 5]. We now present an account of FSTRIPS, a formalism to define such factors, where a *domain theory* expresses assumptions on the structure of states and actions [50, Section 2.1].

A PPI P in Functional STRIPS is defined over a many-sorted first-order logic theory with equality, which we denote as $\mathcal{L}(P) = (T, \Phi, \Pi)$. The constituents of such a theory capture relevant properties of and relations between objects and provide the basic building blocks to factorize states $s \in S$. T is a finite non-empty set of finite sets called *types*, or *sorts*, with a possibly infinite set of variables x_1^t, x_2^t, ... for each type $t \in T$. The *universe* is

the set $U = \cup_{t \in T} t$. Φ is a set of *function symbols* $f \in \Phi$, each of which is said to have a *domain* $\text{Dom}_f \subset t_1 \times \ldots t_i \times \times t_{d_f}$ where $t_i \in T$, and a range $\text{Co}_f \in T$. Π is a set of *relation symbols*. In this work, Π contains the standard relations of arithmetic, i.e., "$<$", "$\leqslant$", "$>$", "$\geqslant$ " in addition to equality "$=$". Some PPIs may specify as well *domain specific* relations, along with their denotation, in addition to the former standard ones. We denote the maximum *function arity* of the domain, $\max_{f \in \Phi} d_f$, as K_f. States $s \in S$ in an FSTRIPS problem P are *semantic structures* that set the interpretation of formulas over $\mathcal{L}(P)$ with fixed universe U. Thus each s contains the *graph* [16] of each function $f \in \Phi$, providing the denotation for *functional terms* $f(\bar{t})$, where $\bar{t} \in \text{Dom}_f$. We note that $S(P)$ is a finite set since types $t \in T$ are finite sets too.

The transition relation $\rightarrow$ is specified via suitably defined *action schemas* $\alpha \in Act$. α is a non-logical symbol such that $\alpha \notin \Phi \cup \Pi$. Actions $\alpha(\bar{x})$ capture sets of transitions in $\rightarrow$ parameterized by a tuple of typed variables $\bar{x} = (x_1^{t_1}, \ldots, x_{d_\alpha}^{t_{d_\alpha}})$. $\tau(\alpha)$ denotes the tuple of types of parameters of α, $\tau(\alpha) = (t_1, \ldots, t_{d_\alpha})$. For each action schema $\alpha(\bar{x})$ we are given $\text{Pre}_\alpha(\bar{x})$, a *precondition* formula over $\mathcal{L}(P)$ and variables $\bar{x}$. In this work we consider a fragment of the formulas considered by [50], defined by the following grammar in Backus-Naur form

$$\text{Pre}_\alpha(\bar{x}) := \top \mid \bigwedge_{i=1}^{card(\text{Pre}_\alpha)} f_i(\bar{y}_i) = z_i \tag{5.1}$$

where $\top$ is the tautology, $card(\text{Pre}_\alpha)$ denotes the *number of equality atoms in the formula* Pre_α, $\bar{y}_i \subset \bar{x}$, and $z_i \in U$. Additionally, we are given an *effect* formula Eff_α of the form

$$\text{Eff}_\alpha(\bar{x}) := \bigwedge_{j=1}^{card(\text{Eff}_\alpha)} f_j(\bar{y}_j) = z_j \tag{5.2}$$

where $card(\text{Eff}_\alpha)$ denotes the *number of equality atoms in the formula* Eff_α, $\bar{y}_j \subset \bar{x}$ and $z_i \in U$.

We now explain the FSTRIPS representation for the *visitall* problem domain from the IPCs [77] in which an agent must visit all the cells in an $n \times n$ square grid starting from the center of the grid. In *Visitall*, we only have one *type*, $T := \{C\}$, where C is a set of cells in the grid, the set of *function symbols* is $\Phi := \{at, visited\}$, and it has a *domain specific* relation $\Pi := \{connected\}$. It has one action schema *move*, which allows the agent to move between two *connected* grid cells. Thus, $Act = \{move\}$. The *move*

schema takes two parameters which we denote by $\bar{x} := (x_1, x_2)$, where x_1 is the current position of the agent and x_2 is the target position. The precondition and effect formulas of *move* are defined as follows

$$\mathrm{Pre}_{move}(\bar{x}) := at() = x_1 \wedge connected(x_1, x_2) = \top \tag{5.3a}$$

$$\mathrm{Eff}_{move}(\bar{x}) := at() = x_2 \wedge visited(x_2) = \top \tag{5.3b}$$

The goal condition of *visitall* requires the agent to visit all cells in C

$$Goal_{move} := \bigwedge_{c \in C} visited(c) = \top \tag{5.4}$$

A set of action schemas Act is *systematic* [98, 129] for a PPI P if and only if, for every $(s, a, s') \in \rightarrow$ there is an action schema $\alpha(\bar{x})$ such that there exists a vector $\bar{v} \in t_1 \times \cdots \times t_{d_\alpha}$ and the following conditions hold:

$$s \models \mathrm{Pre}_\alpha(\bar{x})/\bar{v} \tag{5.5a}$$

$$s' \models \mathrm{Eff}_\alpha(\bar{x})/\bar{v} \tag{5.5b}$$

$$s' \models g(\bar{v}) = w, \text{ when } s \models g(\bar{v}) = w \wedge \nexists w', \mathrm{Eff}_\alpha(\bar{x})/\bar{v} \models g(\bar{v}) = w' \tag{5.5c}$$

where $\varphi(\bar{x})/\bar{v}$ is the (ground) formula that results from replacing every occurrence of $x_i \in \bar{x}$ by that of $v_i \in \bar{v}$. The satisfiability relation in (5.5a)–(5.5c) is defined in a standard way [50]. In other words, a set of action schemas Act is systematic whenever these capture every possible reachability (or accessibility) relation between states s and s'. We note that one action schema $\alpha(\bar{x})$ can satisfy the above for many (s_1, a_1, s'_1), (s_2, a_2, s'_2), ..., each of these tuples providing the semantics of *ground action* $\alpha(\bar{v})$. In this work, we further assume that action schemas do not change the denotation of any symbol in Π (see section 5.5 for a discussion of their treatment in our encoding).

We denote the maximal arity of action schemas in Act as $K_\alpha = \max_{\alpha \in Act} d_\alpha$. The maximal number of equality atoms in precondition formulas is written as $K_{\mathrm{pre}} = \max_{\alpha \in Act} card(\mathrm{Pre}_\alpha)$ (resp. $K_{\mathrm{eff}} = \max_{\alpha \in Act} card(\mathrm{Eff}_\alpha)$ for effects).

Symbol	Description
act_i	Action assigned to slot i, $i = 1, \ldots, N_\pi$
$active_{ik}$	Pin is active
arg_{ij}	Value of j-th argument of slot i, $j = 1, \ldots, K_\alpha$
in_{ik}	k-th input pin data of slot i
out_{il}	l-th output pin data of slot i
spt_{jk}	Output pin supporting k-th input pin of slot j

TABLE 5.1: Quick reference table for the decision variables in the model. N_π is the number of slots, K_α is the (constant) maximal number of arguments in any action schema $\alpha \in Act$.

5.3 Planning as Satisfiability

The approach known as *planning as satisfiability* [84] proceeds by considering a sequence of instances for a related decision problem, that of *plan existence*. We state the latter simply as follows: given some PPI P and parameter N_π, with actions and states defined in terms of some domain theory, the task is to prove that a feasible trajectory σ exists such that length$(\sigma) = N_\pi$, or alternatively, certify that no such σ exists. The classic algorithm for optimal planning in this framework thus considers the sequence of CSPs T_{P,n_0}, T_{P,n_1}, ..., T_{P,n_k}, ... each of these a reduction of the plan existence problem for P and $N_\pi = n_k$ into that of the *satisfiability* of a CSP T_{P,n_k} with suitably defined decision variables and constraints. The sequence of natural numbers $n_0, n_1, \ldots, n_k, \ldots$ is typically, but not necessarily [141], defined as $n_0 = 0$, $n_1 = 1$, and so on. When defined in this manner, as soon as T_{P,n_k} is satisfiable, we have proven that $c^* = n_k$. Scalable certification of infeasibility in this framework has been an open problem until recently [39], yet remains challenging.

5.4 Lifted Causal CP Model

We start by giving a high-level account of our proposed encoding of CSPs T_{P,N_π} and explain the roles played by the decision variables in Table 5.1. Central to our encoding is the notion of plan step or *slot*, of which we have one for *each* action in a plan. In contrast with the causal encoding of KMS, slots are *totally ordered* thus greatly simplifying the definition of persistence that we use to deal with the frame axioms. To each slot *exactly one* action schema $\alpha \in Act$ needs to be assigned (variables act_i), which in turn restricts choices on the possible values for the *arguments* (variables arg_{ij}) of the schema α as

per $\tau(\alpha)$. The assignments to the variables act_i and arg_{ij} determine the choices of *input and output pins* for a slot. A *pin* is a vector of decision variables that we use to represent equality atoms $f(\bar{y}) = z$. These variables choose the function symbol f, terms (constants in U) $\bar{y}$ (a vector) and z. Input pins of slot i thus encode the equality atoms in $\text{Pre}_\alpha(\bar{x})/\bar{v}$ required to be true in state s_{i-1}, and output pins the atoms in $\text{Eff}_\alpha(\bar{x})/\bar{v}$ required to be true in s_i, where $\alpha(\bar{v})$ is the ground action selected by the assigned schema and (possibly partially) assigned values to arguments. These dependencies between the variables of a slot are depicted in Figure 5.1(a), and Figure 5.1(b) illustrates the *active* constraints between variables when the *move* action schema is chosen at slot i in the *visitall* problem domain. The *move* schema has two arguments representing an agent's current and target positions in an $n \times n$ grid. Thus, when $act_i = move$, we require that $arg_{i1} \in C$ and $arg_{i2} \in C$, the input pin in_{i1} is assigned the function symbol *at* and the variable y_{i1} holds an equality relation with arg_{i1}.

We note that we create upfront variables for arguments and pins following from K_α, K_f, K_{pre} and K_{eff}, all constants given by the data in an instance P. For a given slot i, argument variables arg_{ij} that are not used by the schema assigned are set to a special null constant. To turn off pins not needed to represent atoms in preconditions or effect formulas, we have a Boolean variable $active_{ik}$ that indicates if they are being used. Finally, variables spt_{jk} allow choosing what output pin supports a given input pin. These variables allow encoding *causal links* [98] without referring explicitly to atoms.

5.4.1 Variables and Constraints

We now give a formal and precise account of the variables and constraints in the model. Let N_π be the maximal number of slots in a valid plan. For every slot $i = 1, \ldots, N_\pi$ we have integer variables $act_i \in [Act]$[1] to choose the action schema $\alpha \in Act$ assigned to it. Argument variables arg_{ij} select the values of the variables introduced by lifting and their domains correspond to the types $\tau_j(\alpha)$, whenever $act_i = \alpha$

$$(act_i = \alpha) \rightarrow arg_{ij} \in [\tau_j(\alpha)],\ j = 1, \ldots, d_\alpha \tag{5.6}$$

[1] We use the notation $[S]$ to designate the *indexing* of the elements of a set, e.g. $[S] = 1, 2, \ldots$ for $S = \{e_1, e_2, \ldots\}$.

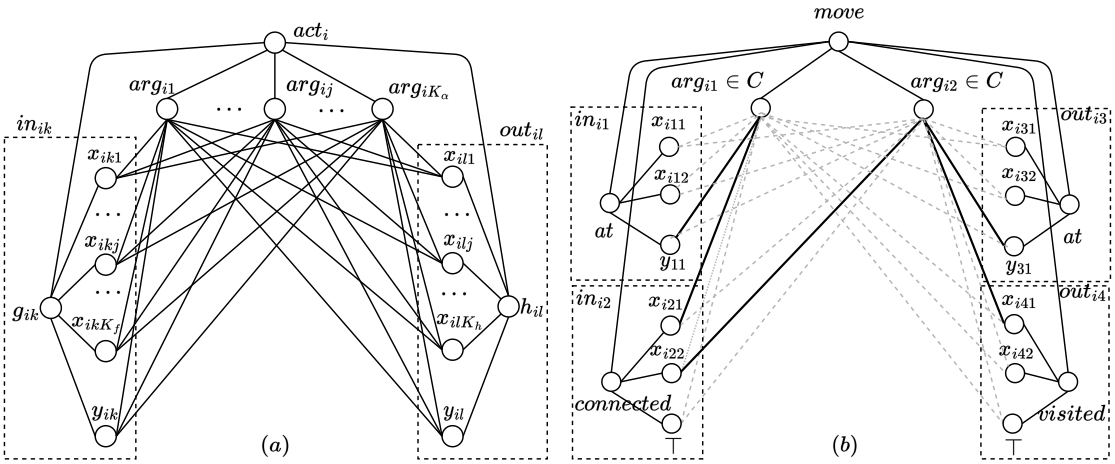


FIGURE 5.1: (*a*) Constraint graph depicting dependencies between the decision variables to model slots (plan steps). There is one vertex for every decision variable, and there is an edge between two variables whenever they appear together in the definition of at least one constraint. (*b*) Active constraints when the *move* action schema is chosen in *visitall*.

The choice of action schema assigned to a slot restricts the choices of (ground) precondition and effect formulas. As introduced above, the input and output pins of a slot i are the vectors of decision variables that select function symbols, arguments, and values $in_{ik} := (g_{ik}, x_{ik1}, \ldots, x_{ik\bar{N}_f}, y_{ik})$ and $out_{il} := (h_{il}, x_{il1}, \ldots, x_{il\bar{N}_f}, y_{il})$. Consistency of schema, preconditions, and effects assigned to slot i is enforced by

$$(act_i = \alpha) \rightarrow (g_{ik} = f_k) \wedge \mathrm{bind}(f_k, \bar{x}_{ik}, y_{ik}, \overline{arg}_i) \tag{5.7a}$$

$$(act_i = \alpha) \rightarrow (h_{il} = f_l) \wedge \mathrm{bind}(f_l, \bar{x}_{il}, y_{il}, \overline{arg}_i) \tag{5.7b}$$

for $k = 1, \ldots, card(\mathrm{Pre}_\alpha)$ and $l = 1, \ldots, card(\mathrm{Eff}_\alpha)$. $\overline{arg}_i$ is the vector of argument variables for slot i. The predicate bind in (5.7a) and (5.7b) ensures that subterms $\bar{x}$ and y of equality atoms in preconditions and effects are consistent with the definition of schema α, expanding into the following

$$\bigwedge_{j=1}^{d_f} \left(\bigvee_{j_2=1}^{K_\alpha} (x_j = arg_{ij_2}) \right) \wedge \bigvee_{j_2=1}^{K_\alpha} (y = arg_{ij_2}) \tag{5.8}$$

The dependencies induced by constraints (5.6)–(5.8) are depicted in Figure 5.1,

States are represented implicitly in our model, and to ensure that no change in the initial state s_0 is allowed without an event in the plan that explains any changes we rely on the notion of causal consistency or *persistence*.

Definition 5.1. Let $f(\bar{x})$ be a functional term $f \in \Phi$ and y a valid value in Co_f. We say that the *truth* of atom $f(\bar{x}) = y$ *persists* between slots i and j, $0 \leqslant i < j \leqslant N_\pi$, if (1) $f : \bar{x} \mapsto_{s_i} y$, and (2) for every slot j', $i < j' < j$, there is no pin $out_{j'l} = (h_{j'l}, \bar{x}_{j'l}, y_{j'l})$ such that $h_{j'l} = f$, $\bar{x}_{j'l} = \bar{x}$, and $y_{j'l} \neq y$.

This ensures that states s_j are given either by 1) function and value assignments in the initial state s_0 that have *persisted* through (ground) actions a_i encoded in slots i, $0 < i < j$, or 2) an assignment made by some action a_i encoded in some slot i, $0 < i < j$, that has persisted through the actions $a_{j'}$ encoded by slots j', $i < j' < j$. For example, in *visitall*, an atom $at() = c_1$ in the initial state, where c_1 is the initial position of the agent, is said to have *persisted* until slot 3 iff the atom *holds* at slot 1 and 2.

The many possible cause-and-effect relations between output and input pins that may justify the causal consistency of a plan are modeled via so-called (causal) *support* variables, $spt_{jk} \in \big([0, j-1] \times [1, K_{\text{eff}}]\big) \cup \{\text{null}\}$, for each slot $0 < j \leqslant N_\pi$ and input pin $1 \leqslant k \leqslant K_{\text{pre}}$. The domain of these variables is the set of two-dimensional vectors whose first element is the index of the slot i and the second the index l of the output pin supporting in_{jk}, plus a dummy vector null that indicates that input pin in_{jk} does not have causal support assigned. The following constraints enforce that every active pin has a matching supporting one

$$spt_{jk} = \text{null} \rightarrow \neg active_{jk} \tag{5.9a}$$

$$spt_{jk} = (i, l) \rightarrow out_{il} = in_{jk} \tag{5.9b}$$

$$spt_{jk} = (i, l) \rightarrow \bigwedge_{i<j'<j} \text{persists}(out_{j'l}, in_{jk}) \tag{5.9c}$$

for each $1 \leqslant j \leqslant N_\pi$ and $0 \leqslant i < j$. Constraint (5.9a) excuses input pins that are inactive from having a causally supporting output pin. Constraint (5.9b) ensures that the values for functions, domain, and valuation set by pins in_{jk} and out_{il} are matching. Constraint (5.9c) encodes the requirement of causal supports to not be interfered with by any ground action set for intermediate slots $i < j' < j$. $\text{persists}(out_{il}, in_{jk})$ in constraint (5.9c) expands into the formula

$$(h_{il} \neq g_{jk}) \vee (\bar{x}_{il} \neq \bar{x}_{jk}) \vee (y_{il} = y_{jk}) \tag{5.10}$$

In the specific case of *visitall*, the constraints (5.9) and (5.10) impose additional constraints on the input and output pin variables than the ones depicted in Figure 5.1(b), thus, restricting the choices of act_i and $\overline{arg}_i$ as well. For example, if $spt_{31} = (1, 3)$, then the constraint (5.9b) requires that the assignment to $out_{13} := (at, y_{13})$ matches that of $in_{31} := (at, y_{31})$, i.e., $y_{13} = y_{31}$, and the constraints (5.9c) and (5.10) ensure that the assignment to out_{13} *persists* through slot 2, i.e., the (ground) action a_2 does not affect the interpretation of at. It is easy to see that when $spt_{31} = (1, 3)$, the choice of y_{13} in turn affects arg_{31} since arg_{31} holds an equality relation with y_{31}.

Additionally, whenever we assign action schemas α to slots i we mark input and output pins as being active

$$act_i = \alpha \rightarrow active_{ik},\ act_i = \alpha \rightarrow \neg active_{ik'} \tag{5.11a}$$

$$act_i = \alpha \rightarrow active_{il},\ act_i = \alpha \rightarrow \neg active_{il'} \tag{5.11b}$$

with indices ranging as follows: $1 \leqslant k \leqslant card(\mathrm{Pre}_\alpha)$, $card(\mathrm{Pre}_\alpha) < k' \leqslant K_{\mathrm{pre}}$, $1 \leqslant l \leqslant card(\mathrm{Eff}_\alpha)$, $card(\mathrm{Eff}_\alpha) < l' \leqslant K_{\mathrm{eff}}$.

Initial and goal states are accounted for in the following way. To model the initial state, we define slot 0 to consist exclusively of a set of (output) pins out_{0l} where l ranges over the indexing of the set

$$\mathcal{F} := \bigcup_{f \in \Phi} \mathrm{Dom}_f \tag{5.12}$$

and each pin is set as per constraints, for each $l \in [\mathcal{F}]$

$$out_{0l} = (f_l, \bar{x}_l, y) \tag{5.13}$$

and $y \in \mathrm{Co}_f$ such that $f_l : \bar{x}_l \mapsto_{s_0} y$. Goal states are modeled too by having the slot N_π have a special structure, in this case by only having input pins $in_{N_\pi k}$ with k ranging over the indexing of the set of equality atoms $\varphi_k \equiv f_k(\bar{x}_k) = y_k$ in the goal formula

$$\bigwedge_{\varphi_k \in Goal} in_{N_\pi k} = (f_k, \bar{x}_k, y_k) \wedge active_{N_\pi k} \tag{5.14}$$

5.4.2 Analysis: Systematicity and Complexity

We first establish as fact that our encoding is *systematic* [98].

Theorem 5.2 (Systematicity)**.** *Let $T_{P,N_\pi} = (\mathcal{X}, \mathcal{C})$ be the CSP given by the decision variables $\mathcal{X}$ in Table 5.1 and set of constraints $\mathcal{C}$ (5.6)–(5.14), for some suitable choice of N_π. There exists an assignment ξ onto variables $\mathcal{X}$ that satisfies every constraint in $\mathcal{C}$ if and only if there exists a feasible solution σ to the PPI P, whose actions are given by slots, argument, input, and output pin variables values.*

Proof. It follows trivially from the definitions given in the previous sections that assignments ξ encode finite trajectories $\sigma = s_0 a_1 s_1 \dots s_{N_\pi}$. To prove the forward direction, it suffices to observe that (1) constraints (5) on input and output pins for a slot i define *implicitly* sets of pairs of states $(s_{i-1}, s_i) \in \rightarrow_{act_i}$, the set of transitions corresponding to the schema $act_i = \alpha$ assigned to the slot, (2) constraints (11) ensure that for $i = 1$ the predecessor of s_1 corresponds with the initial state s_0 given in the definition of the PPI P, and (3) the last state in the trajectory given by ξ the equality atoms in *Goal*. To prove the reverse direction, we note that each pair of consecutive states s_{i-1} and s_i must belong to exactly one of the transition sets $\rightarrow_\alpha$. From the definition of this set, the schema, argument, and pins assignments follow directly. □

Giving bounds on the number of variables generated for a CP model is not as informative as doing so for CNF formulas due to the advanced *pre-solving techniques* that state-of-the-art CP solvers employ [113], that may introduce auxiliary Boolean variables and eliminate variables whose values can be determined without searching. Assuming that no such transformations are applied to the CSP, the set of spt_{jk} variables and their domain constraints, which are required to be encoded explicitly by LCG solvers [108], is the largest and is $O(N_\pi^2 K_{max})$ where $K_{max} = \max\{ K_{\text{pre}}, K_{\text{eff}} \}$. In the IPC benchmarks, this often results in just a quadratic rate of growth as K_{max} is usually much smaller than N_π for optimal plans. Yet, as we will see in our Evaluation, this is not always the case, and we get cubic rates.

5.5 Programming the Model with CpSat

To implement the CP model introduced in the previous section, we have used the LCG solver bundled in Google's Or-Tools package, CpSat [113]. At the time of writing this, CpSat is the state-of-the-art LCG solver as adjudicated by the latest results of the Minizinc challenge [142]. A key feature of CpSat we rely on is the extensive support for different variants of so-called element$(\cdot)$ constraints [74]. These constraints implement *variable indexing*, a key modeling feature in CP, and correspond with the statement $\bar{v}_x = z$, that reads as "the element at position x of vector $\bar{v}$ must be equal to z". The power of these constraints lies in the possibility of elements of $\bar{v} = (v_1, \ldots, v_m)$, x and z being *all* decision variables. We write element$(\cdot)$ constraints using Hooker's [74] notation

$$\text{element}(x, z \mid \bar{v}) \tag{5.15a}$$

$$\text{element}((x_1, x_2), z \mid V) \tag{5.15b}$$

where (5.15a) implements the statement $\bar{v}_x = z$, and (5.15b) implements $V(x_1, x_2) = z$ that reads as "the element of matrix V at coordinates x_1, x_2 must be equal to z". x, x_1 and x_2 are thus *index* variables that locate one decision variable within a collection.

In our implementation, the action assigned to slot i, act_i is an index variable, which depends on the data assigned to input and output pins as per constraints (5.7) and (5.8). Both of these constraints can be encoded compactly using (5.15a)

$$\text{element}(act_i, g_{ik} \mid (f_1, f_2, \ldots, f_j, \ldots, f_{|Act|})) \tag{5.16a}$$

$$\text{element}(act_i, x_{ikl} \mid (\nu_1, \nu_2, \ldots, \nu_j \ldots, \nu_{|Act|})) \tag{5.16b}$$

$$\text{element}(act_i, y_{ik} \mid (\mu_1, \mu_2, \ldots, \mu_j \ldots, \mu_{|Act|})) \tag{5.16c}$$

Where f_j is a function symbol, ν_j and μ_j are the argument variables of slot i or constants consistent with the definition of action schema when $act_i = j$. f_j is assigned a special function when the input pin is *inactive* in the action schema, thus accounting for literals $\neg active_{ik}$.

To give the readers a better intuition of the element constraints above, we revisit the example of *visitall*. Visitall only has one action schema *move*, and hence, the element

constraints are uncomplicated. Consider the input pin in_{i1} in Figure (5.1)(b), the constraint (5.16a) for the pin is written as element$(act_i, g_{i1} \mid (at))$, the constraint 5.16b as element$(act_i, x_{i1l} \mid (\square))$, and lastly the constraint (5.16c) as element$(act_i, y_{i1} \mid (arg_{i1}))$, where $\square$ is a *dummy* symbol which accounts for *inactive* variables These constraints then ensure that the variables of the slot i are internally consistent for all possible assignments to act_i.

The *support* variables spt_{jk} are index variables with two elements, (i, l), the first being the index of a slot and the second the index of an output pin. As discussed in the previous section, variables spt_{jk} depend on the data of input pin in_{jk}. Output pins out_{jl} and constraints (5.9a) and (5.9b) can be accounted for using element$(\cdot)$ in its matrix form (5.15b)

$$\text{element}((i, l), in_{jk} \mid H) \tag{5.17}$$

where H is the data of all output pins in a 2-dimensional grid. Constraint (5.17) thus requires that the data of the output pin at the row i (slot i) and column l (l-th pin) of H is equal to that in in_{jk}. The matrix H includes a specially defined element, containing the data used to represent *inactive* pins, whose index is assigned to spt_{jk} when the k-th input pin at slot j is *inactive*, thereby encoding constraint (5.9a).

We exploit other modeling features offered by CpSat to implement the persist$(\cdot)$ predicate given in Equation (5.10)

$$u \vee v_1 \vee \ldots \vee v_o \vee \ldots \vee v_{d_f} \vee w \tag{5.18a}$$

$$u \rightarrow (h_{il} \neq g_{jk}) \tag{5.18b}$$

$$v_o \rightarrow (x_{ilo} \neq x_{jko}) \tag{5.18c}$$

$$w \rightarrow (y_{il} = y_{jk}) \tag{5.18d}$$

where u, v_o and w are auxiliary Boolean variables created for each $(out_{j'l}, in_{jk})$ pair, and $1 \leqslant o \leqslant K_f$. CpSat does not represent explicitly constraints (5.18b), (5.18c) and (5.18d). Instead, it collects them into a *precedences propagator* as *inequalities* between integer variables. The *precedences propagator* uses the Bellman-Ford algorithm to detect negative cycles in the constraint graph of inequalities, and propagates bounds on the integer variables [105]. Still, $O(N_\pi^2 K_{\text{pre}} K_{\text{eff}} K_f)$ variables are generated in the worst-case

scenario, e.g., $O(n^5)$ if all these quantities belong to the same order of magnitude. In most of the benchmarks we use to test planners using these encodings, the number of preconditions, effects and arity of functions are much smaller than N_π, thus generating $O(n^2)$ variables.

Encoding *static* relations with *table constraints*. Some PPIs contain *domain specific* relations unaffected by the action schemas. For example, in *visitall*, the relation *connected* is *static*, and the specification of the initial state fixes its interpretation. We encode the dependency of the input pin variables on these static relations using *table constraints*, which allow us to specify a set of *allowed* (or *forbidden*) assignments to a tuple of variables, i.e., for a static relation R, we encode the constraint as $table(in_{ik}, R)$, where the pin in_{ik} is specially designed to represent a tuple in R. This is more efficient than using the element constraints (5.17) since CpSat translates them into a concise CNF formulation.

Propagator for Required Persistence. We recall constraint (5.9c) that enforces the requirement that whenever an output pin out_{il} is to provide a causal explanation for an input pin in_{jk}, the atom described by the former is not interfered with by any other output pins in intermediate slots. Clearly, the number of variables and clauses (5.18a) is proportional to $j - i$, bringing the potential number of variables generated to be $O(n^6)$. While such a rate of growth seems unsustainable for non-trivial instances, the empirical results we obtain clearly show that this worst-case does not always follow for many domains widely used to evaluate planning algorithms.

To avoid this blow-up, yet keep the strong inference offered by the system of constraints (5.18), we introduce a specific propagator that interfaces with CpSat *precedences* propagator and checks whether constraint (5.9c) is satisfied. The propagator activates whenever an assignment in the CDCL search fully decides the variables in input pin in_{jk}. We then check, for every slot j', $0 < j' < j$, whether the current assignment has fully decided some output pin $out_{j'l}$, and proceed to evaluate the persistence predicate in Equation (5.9) on the assignment. If the former evaluates to false, we have a conflict between the assignment and constraint (5.9c). We then generate the following blocking clause or *reason* to explain it

$$\neg(\varphi_{out_{j'l}} \wedge \varphi_{in_{jk}}) \vee \big(LB(spt_{jk_1}) > j'\big) \tag{5.19}$$

where φ formulae are the conjunction of equality atoms that bind variables in pins to the values in the current assignment. spt_{jk_1} is the first element (variable) in spt_{jk} and LB is a function provided by CpSat that allows access to the lower bound of the domain of a variable in constant time.

Searching for plans. Our algorithm for planning as satisfiability uses a strategy to find plans that are analogous to the notion of *lookaheads* in Approximate Dynamic Programming. As in the classic algorithm described earlier, we generate a sequence of CSPs $T_{P,k_1}, T_{P,k_2}, \ldots, T_{P,k_i}, \ldots$ with $k_0 = 0$ and $k_i - k_{i-1} \geqslant 1$ for $i > 0$.

To each T_{P,k_i} we impose the Pseudo-Boolean objective

$$\min \sum_{j=1}^{k_i} enabled_j \tag{5.20}$$

where $enabled_j$ are auxiliary Boolean variables which we use to "switch off" slots via constraints $\neg enabled_j \rightarrow act_j > |Act|$. If CpSat proves that the resulting *optimization* problem has *finite* optimal value z then we know that $c^* = z$ and the search terminates as we have found an optimal plan. If CpSat finds an upper bound z, that is, a sequence of feasible solutions is found, but it is not possible to prove the optimality of the last one within the time limit set, then we know that $c^* \leqslant z$. If CpSat proves the tightest upper bound to be ∞ (e.g., the problem is unsatisfiable), then we have established a *deductive lower bound* [56], as we know that $c^* > k_i$. In this last case, we repeat the process above with $TP_{k_{i+1}}$ until a solution is found or the allowed time to search for plans is exhausted.

5.6 Functional transformation of a PDDL task

Benchmarks in planning are not expressed in FSTRIPS directly but in PDDL [66], a defacto abstract representation of a PPI used by the research community. PDDL is defined by the tuple $P = \langle U, T, \mathcal{P}, Act, s_0, \gamma \rangle$, where U is a set of *objects*, $T \in 2^U$ is a collection of types, $\mathcal{P}$ is a set of *predicates*, Act is a set of *action schemas*, s_0 is the initial state, and γ is the *goal formula*. For a given predicate $\mathsf{P} \in \mathcal{P}$ of arity d_{P}, there are two associated literals, the positive atom $\mathsf{P}(\bar{x})$ and negative atom $\neg\mathsf{P}(\bar{x})$, where $\bar{x}$ is tuple of variables $(x_1, x_2, \ldots, x_{d_{\mathsf{P}}})$, the domain of x_i corresponds to a type $t_i \in T$. We denote $\max_{\mathsf{P} \in \mathcal{P}} d_{\mathsf{P}}$ as K_{P}. A *simple* reformulation of PDDL planning task into FSTRIPS representation is to treat each predicate $\mathsf{P} \in \mathcal{P}$ as a Boolean function or a *mapping*,

$f_{\mathsf{P}} : \mathrm{Dom}_{f_{\mathsf{P}}} \mapsto \mathrm{Co}_{f_{\mathsf{P}}}$, $\mathrm{Dom}_{f_{\mathsf{P}}} \subseteq t_1 \times t_2 \times, \ldots, \times t_{d_{\mathsf{P}}}$, $\mathrm{Co}_{f_{\mathsf{P}}} = \{\top, \bot\}$. Thus, $(f_{\mathsf{P}}(\bar{x}) = \top)$ denotes $\mathsf{P}(\bar{x})$ and $(f_{\mathsf{P}}(\bar{x}) = \bot)$ denotes $\neg\mathsf{P}(\bar{x})$. Mappings beyond Boolean functions can yield a more compact FSTRIPS representation for CP. Hence, in this section, we present a novel method to derive a more concise functional transformation of predicates $\mathcal{P}$.

Any function $f : \mathrm{Dom}_f \mapsto \mathrm{Co}_f$ has an associated binary relation, a *mapping*, $\mathcal{R}_f := \{(x, y) \mid f(x) = y,\ x \in \mathrm{Dom}_f, y \in \mathrm{Co}_f\}$. A predicate $\mathsf{P} \in \mathcal{P}$ of arity 2 also has an associated binary relation, $\mathcal{R}_{\mathsf{P}} := \{(x, y) \mid \mathsf{P}(x, y), x \in \mathrm{Dom}_f, y \in \mathrm{Co}_f\}$. If the relation $\mathcal{R}_{\mathsf{P}}$ is a mapping, then we can create a *more concise* transformation, i.e. $f_{\mathsf{P}} : \mathrm{Dom}_f \mapsto \mathrm{Co}_f$, instead of $f_{\mathsf{P}} : \mathrm{Dom}_f \times \mathrm{Co}_f \mapsto \{\top, \bot\}$. Moreover, 0-ary, 1-ary, and n-ary predicates can all be mapped into the binary case without loss of generality, i.e., $\mathsf{P}()$ can be substituted by $\mathsf{P}(c_1, c_2)$, $c_1, c_2 \in \mathrm{Dom}_c, \mathrm{Dom}_c \cap U = \varnothing$, $\mathsf{P}(y)$ by $\mathsf{P}(c, y)$, and for $n \geqslant 2$, $\mathsf{P}(x, y)$ replaces the predicate $\mathsf{P}(u_1, \ldots, u_n)$, where x is a tuple in the set of combinations of parameters of length $n-1$ and y is the parameter which is excluded from x. For example, in the PDDL specification of *visitall*, *at* is a 1-ary predicate representing the agent's current position. We can map *at* into the binary case by introducing a constant A, then transform it into a function as $at : \{A\} \mapsto C$ since the agent can take at most one position on the grid. Thus, for each predicate $\mathsf{P} \in \mathcal{P}$, there is an associated binary relation $\mathcal{R}_{\mathsf{P}} := \{(x, y) \mid \mathsf{P}(x, y)\}$. There are two *necessary and sufficient* conditions for a binary relation to be a *mapping*, (1) it is *right-unique*, and (2) it is *left-total*. A binary relation $\mathcal{R}_{\mathsf{P}}$ is *right unique* iff $\forall\ x_1, x_2 \in \mathrm{Dom}_x, y_1, y_2 \in \mathrm{Dom}_y,\ \mathsf{P}(x_1, y_1) \wedge \mathsf{P}(x_2, y_2) \wedge (x_1 = x_2) \rightarrow (y_1 = y_2)$. It is *left-total* iff $\forall\ x \in \mathrm{Dom}_x,\ \exists\ y \in \mathrm{Dom}_y,\ \mathsf{P}(x, y)$. Since the states $s \in S$ set the interpretation of predicate P, we have to at least prove that the *right-unique* and *left-total* conditions *hold* in S^R_P, the set of states reachable from s_0, to make a case for the *more concise* functional transformation.

Theorem 5.3. *If $\mathcal{R}_{\mathsf{P}}$ is a mapping in all possible interpretations $s \in S^R_P$, then P', the transformation of the problem P which encodes the predicate P as a function $f_{\mathsf{P}} : \mathrm{Dom}_x \mapsto \mathrm{Dom}_y$, has the same set of reachable states as P, i.e. $S^R_P = S^R_{P'}$*

Proof. If the *right-unique* and *left-total* properties *hold* for $\mathcal{R}_P$ in all $s \in S^R_\Pi$, then applying the functional transformation would not alter the reachable state space since the functional transformation *implicitly* enforces the same conditions, i.e. $\forall\ x_1, x_2 \in \mathrm{Dom}_x, y_1, y_2 \in \mathrm{Dom}_y,\ (f_P(x_1) = y_1) \wedge (f_P(x_2) = y_2) \wedge (x_1 = x_2) \rightarrow (y_1 = y_2)$ and $\forall\ x \in \mathrm{Dom}_x,\ \exists\ y \in \mathrm{Dom}_y,\ (f_P(x) = y)$. □

A *sufficient* condition for the *right-unique* property to hold in all interpretations $s \in S_P^R$ is that the *negation* of the *right-unique* condition is *false* in $S_{P_r}^R \supseteq S_P^R$, where P_r is a *relaxation* of P. Thus, we can do a *relaxed* reachability analysis of the formula $\psi_{\mathsf{P}} := \exists x_1, x_2, y_1, y_2,\ \mathsf{P}(x_1, y_1) \wedge \mathsf{P}(x_2, y_2) \wedge (x_1 = x_2) \wedge (y_1 \neq y_2)$ to test whether the *right-unique* condition holds for all $s \in S_{P_r}^R$, i.e. the *right-unique* property *holds* if ψ_{P} is unreachable in P_r. The *relaxation* is important since checking the reachability of the condition in P is as complex as solving the problem itself. To this end, we extend the h^m heuristic [58], which is an *admissible approximation* of the optimal heuristic function h^*, to the first-order logic existential formula of the form $\psi := \exists \bar{x}, \psi^L \wedge \psi^{EQ}$, where $\bar{x} := (x_1, x_2, \ldots, x_n)$ is a vector of parameters, ψ^L is a conjunction of literals whose interpretation is set by the states $s \in S_P^R$, and ψ^{EQ} is an equality-logic formula. We then use the extension of h^m to test the reachability of the formula ψ_{P}, i.e., if $h^m(\psi_{\mathsf{P}}) = \infty$, then ψ_{P} is unreachable, and hence, the right unique condition holds in all interpretations $s \in S_P^R$ of P. Also, we note that, if $\mathcal{R}_{\mathsf{P}}$ is *right-unique*, the *left-total* property is trivial to satisfy. For each $x \in \mathrm{Dom}_x$, if $\nexists y \in \mathrm{Dom}_y,\ \mathsf{P}(x, y)$, then we map x to $\square$, a *dummy* constant symbol.

5.7 h^m heuristic over first-order logic existential formulae

In this section, we present an account of the h^m *admissible* heuristic [58] and its extension to the first-order logic existential formulas, which we then use to test the reachability of a first-order logic formula. The heuristic function h^m is an *admissible approximation* of the optimal heuristic function $h^* : \mathcal{A} \mapsto \mathbb{N}$, defined over a conjunction of positive and negative *ground* atoms $\mathcal{A} := \mathcal{P} \times t_1 \times \ldots \times t_{K_{\mathcal{P}}} \times \{\top, \bot\}$. h^* is defined using the *regression* model of the planning problem, in which we regress *backward* from a *goal* formula using a *regression function* $rg_a : \mathcal{A} \mapsto \mathcal{A}$ with respect to a ground action $a \in Act \times U^{K_\alpha}$ [122]. We extend the *regression* function rg_a to the first-order logic existential formulas and use it to define h^m for first-order logic formulas.

Let ψ be a first-order logic existential formula, $\psi := \exists \bar{x}, \psi^L \wedge \psi^{EQ}$, where $\bar{x} := (x_1, x_2, \ldots, x_n)$ is a vector of parameters, ψ^L is a conjunction of literals whose interpretation is set by the states $s \in S_P^R$, and ψ^{EQ} is an equality-logic formula. We denote the set of literals in a formula ϕ by $lits_\phi$, the predicate and the argument variables of a literal $l \in lits_\phi$ by $predicate_l$ and $\overline{arg}_l$, respectively, and the polarity of l by $polarity_l$.

The *regression* of formula ψ with respect to an action schema $\alpha := \langle \text{Pre}_\alpha, \text{Eff}_\alpha \rangle$ involves identifying the *supporter-supportee* pairs and the *inconsistent* literal pairs between Eff_α and ψ^L. A *supporter-supportee* relationship holds between $l \in lits_{\text{Eff}_\alpha}$ and $l' \in lits_{\psi^L}$ iff the predicate and the arguments of l *match* that of l' and they have the *same polarity*. On the other hand, a literal $l \in lits_{\text{Eff}_\alpha}$ is *inconsistent* with $l' \in lits_{\psi^L}$ iff the predicate and the arguments of l *match* that of l' but they have the *opposite polarity*.

While *regressing* with respect to action schema α, we need to consider every possible combination of *supporter-supportee* pairs, i.e. all subsets of the set of *potential* supporter-supportee pairs $SP := \{(l, l') \mid polarity_l = polarity_{l'},\ predicate_l = predicate_{l'},\ l \in lits_{\text{Eff}_\alpha}, l' \in lits_{\psi^L}\}$. Then for each literal pair (l, l') in the set we add the constraint $\bigwedge_{i=1}^{K_\mathcal{P}} arg_{li} = arg_{l'i}$ to the formula obtained through regression. Thus, the *regression* of ψ using α produces a set of formulas. Similarly, for each pair in the set of *potential* inconsistent pairs $IP := \{(l, l') \mid polarity_l \neq polarity_{l'},\ predicate_l = predicate_{l'},\ l \in lits_{\text{Eff}_\alpha}, l' \in lits_{\psi^L}\}$, we add the constraint $\bigvee_{i=1}^{K_\mathcal{P}} arg_{li} \neq arg_{l'i}$ to the regression formula of ψ. We now present the definition of the *regression function* over first-order logic formulas.

$$rg_\alpha(\psi) := \{\exists \bar{x},\ \hat{rg}_\alpha(\psi^L, \widetilde{SP}) \wedge \check{rg}_\alpha(\psi^{EQ}, \widetilde{SP}) \mid \widetilde{SP} \subseteq SP\} \tag{5.21a}$$

$$\hat{rg}_\alpha(\psi^L, \widetilde{SP}) := \text{Pre}_\alpha \wedge \bigwedge_{l \in \{lits_{\psi^L} \setminus \{l' \mid (l,l') \in \widetilde{SP}\}\}} l \tag{5.21b}$$

$$\check{rg}_\alpha(\psi^{EQ}, \widetilde{SP}) := \psi'^{EQ} \wedge \bigwedge_{(l,l') \in \widetilde{SP}} \wedge_{i=1}^{K_\mathcal{P}} arg_{li} = arg_{l'i} \wedge \bigwedge_{(l,l') \in IP} \vee_{i=1}^{K_\mathcal{P}} arg_{li} \neq arg_{l'i} \tag{5.21c}$$

where we obtain the regression of ψ^L by removing the *supportees* in the set $\widetilde{SP}$ from ψ^L and adding the precondition of α. The regression of ψ^{EQ} involves a reduction into a *canonical* form ψ'^{EQ} by setting the equality atoms whose arguments do not appear in the arguments of the regression $\hat{rg}_\alpha(\psi^L, \widetilde{SP})$ to *true*. Then, we add two equality logic formulas, the first of which *binds* the arguments of *supporter-supportee* pairs in $\widetilde{SP}$ and the second disallows *inconsistent* assignments to the arguments of literals pairs in *IP*.

The h^m heuristics for $\psi := \exists \bar{x}, \psi^L \wedge \psi^{EQ}$ is defined using regression as follows

$$h^m(\psi) := \begin{cases} 0, & s_0 \models \psi \\ min_{\psi' \in rg_\alpha(\psi), \alpha \in Act} h^m(\psi') + cost(\alpha), & |lits(\psi^L)| \leqslant m \\ max_{\psi \vdash \psi', |lits(\psi'^L)| \leqslant m} h^m(\psi'), & otherwise \end{cases} \tag{5.22}$$

$h^m(\psi_\mathsf{P})$ can be efficiently computed using dynamic programming with *memoization*, and $s_0 \models \psi$ can be encoded as a CP program with *equality and table constraints*. For a fixed value of m, the complexity of the above procedure is low polynomial in the number of nodes, i.e., the number of first-order logic formulas $O(|\mathcal{P}|^m \cdot 2^{K_\mathcal{P}})$. An important property of the above procedure is that it is *entirely lifted*, i.e., no *ground* atom would occur in the formulas obtained through regression if no action schema has *ground* atoms. This is especially useful in *hard-to-ground*(HTG) domains where the size of *ground* theory renders the *translation* methods [67] used by most planners intractable.

5.8 Empirical Analysis

Our experiments consist of running a given planner on a PPI, ensuring that the solver process runs on a single CPU core (Intel Xeon running at 2GHz). We impose resource usage limits both on runtime (1800 s) and memory (8 GBytes). We used the *Downward Lab* module [134] to manage the parallel execution of the experiments.

We compare the performance of CpSat solving our model, with and without propagators for persistence constraints, with that of notable optimal and satisficing domain-independent planners. The former include, in no particular order, *lmcut* [69], *symbolic-bidirectional(sbd)* [145], the baseline at the optimal track of the International Planning Competition (IPC) 2018, *cpddl* [46], a very efficient implementation of symbolic dynamic programming and many other pre-solving techniques that analyze the structure of actions in the instance, *delfi1*, a *portfolio* solver [82] and winner of the optimal planning track in IPC 2018, and *lisat* [76], a recently proposed lifted planner which has state-of-the-art performance on *hard-to-ground* (HTG) benchmark tasks. It solves an encoding of lifted classical planning in propositional logic using a highly efficient SAT solver Kissat [10] written in *C*. Satisficing planners include the *satisficing* variant of *lisat*, *Madagascar* [126], a SAT planner which was the runner-up in the Agile track in IPC 2014, and *lifted*

implementations of BFWS planners, BFWS($[\mathrm{R_X}, h^{\mathrm{add}}]$) and BFWS($[\mathrm{R_X}, h^{\mathrm{ff}}]$) [29] which have state-of-the-art *satisficing* performance on HTG benchmark tasks [29, 89]. We use $\mathrm{PL}^{\mathrm{add}}_{R_x}$ and $\mathrm{PL}^{\mathrm{ff}}_{R_x}$ to denote the *lifted* implementations of BFWS planners, and *lisat* and $\overline{lisat}$ to denote the *satisficing* implementation of *lisat* with and without *londex* [24] constraints. All *optimal* planners were configured to minimize the plan length.

We evaluate all planners on the HTG benchmarks and the instances from the *optimal track* of the IPC [77]. Testing the planners on the HTG instances is significant as the size of U is very large, and as a result, explicit grounding is either infeasible or greatly stresses the implementation of key techniques (match trees, compilation into finite-domain representations) [67] that heuristic search planners rely on to be competitive. Comparing the performance with IPC instances allows us to control for implementation-dependent factors and see how CpSat copes with quickly growing numbers of variables and constraints. This is so because instances in the IPC benchmark tend to require a significantly higher number of plan steps for some domains (like *logistics*). We also tested a 3-action lifted formulation of *blocksworld* where actions are *move(x,y,z), move-to-table(x,y), and move-from-table(x,y)*, which we think is significant as it measures the sensitivity of planners to long-studied formulations of the same domain.

In addition to the IPC instances, we evaluate the planners on the multi-modal project scheduling problems (*MMPSP*) from the 'j10' set in PSPLIB [85]. The scheduling benchmarks are particularly interesting to us since they exercise a different combinatorial structure than the IPC instances. While the IPC instances have smaller and non-numeric *sorts*, the scheduling instances usually have numeric duration and resources. To test the sensitivity of the planners to the distinguishing features of scheduling problems, we performed an ablation study by scaling up the duration of the jobs in *MMPSP* by a factor of 2, 8, and 16, respectively.

CpSat Hyperparameters. CpSat offers great flexibility to configure what pre-solving techniques, restarting policies, and branching heuristics are used. In our experiments, we used the default branching heuristic settings and chose the *luby* policy for managing restarts. CpSat default branching heuristic settings try first to fix values of literals appearing in CNF clauses (which may have been lazily generated by some propagator), selecting the former with classical activity-based variable selection heuristics [113]. Integer variables are only considered after all Boolean literals are fixed. We use the *linear scan* algorithm of Perron et al. [113] to optimize Eq. (5.20).

Implementations of Lifted Causal Encodings. We have tested two implementations of constraints (5.9c). The first uses the formulation in Eqs. 5.18. The second one uses the *persistence propagator*. We will refer to them as CP_0 and CP_{PP}, respectively. We set k_1 to goal count for the CSP T_{P,k_1} in our experiments, then used a *luby* sequence to set k_i for $i > 1$. We use the same strategy for *satisficing* planning except we scale the *luby* sequence by a factor of 5 and allocate a time-budget B_i for the CSP T_{P,k_i}. B_i is set such that the planner has sufficient time to explore a *sliding window* over plan-lengths, $B_i := \min\{r,\ r \cdot (k_i - lb_{i-1})/W\}$, where r is the total remaining time-budget, lb_{i-1} is the lower bound on the plan-length returned by the CSP $T_{P,k_{i-1}}$, and W is the size of the *sliding window* which is a planner parameter. We set $W = 50$ in our experiments.

To assess the effectiveness of the functional transformation, we generated the functional representation using the method described at the end of the previous section. We refer to the encoding using functional representation as CP_0^{fn} and $\text{CP}_{\text{PP}}^{fn}$.

Hard-to-ground	Optimal Solution									Satisficing Solution						
Domain	CP_0	CP_0^{fn}	CP_{PP}	$\text{CP}_{\text{PP}}^{fn}$	*lisat*	*lmcut*	*sbd*	*cpddl*	*delfi1*	CP_0^{fn}	$\text{CP}_{\text{PP}}^{fn}$	*lisat*	$\overline{\textit{lisat}}$	MpC	$\text{PL}_{R_x}^{\text{add}}$	$\text{PL}_{R_x}^{\text{ff}}$
blocks-3ops(40)	40	40	40	40	40	0	0	0	0	40	40	40	12	0	10	10
blocks-4ops(40)	40	40	40	40	40	10	0	1	0	40	40	40	20	4	19	17
childsnack(144)	48	48	48	48	49	7	73	81	58	144	144	144	144	66	94	96
ged(156)	23	30	22	31	35	18	12	14	18	37	29	58	28	30	156	156
ged-split(156)	22	24	22	24	26	18	22	30	22	40	38	46	28	150	154	153
logistics(40)	27	35	22	36	28	7	12	0	8	40	40	40	0	0	40	40
org-syn-MIT(18)	18	18	18	18	18	2	2	13	2	18	18	18	10	0	18	18
org-syn-alk(18)	18	18	18	18	18	18	18	18	18	18	18	18	18	0	18	18
org-syn-orig(20)	13	13	15	15	20	0	1	2	1	5	9	14	1	0	13	13
pipes-tkg(50)	15	15	15	17	20	8	12	13	10	23	25	10	23	10	45	46
rovers(40)	3	7	3	8	3	7	2	0	2	11	10	4	0	0	39	40
visitall3D(60)	25	25	24	34	35	33	12	12	24	33	49	46	39	12	57	57
visitall4D(60)	23	23	23	35	34	16	6	6	6	30	44	48	36	6	42	41
visitall5D(60)	27	26	26	33	33	11	0	0	0	32	44	54	38	0	40	40
Total(902)	342	362	336	397	399	155	172	190	169	511	548	580	397	278	745	745

TABLE 5.2: Coverage of planners on *hard-to-ground* benchmark domains.

Performance over benchmarks We now discuss the observed performance (see Table 5.2)measured as *coverage* or the number of instances solved optimally (or sub-optimally) per problem domain in each benchmark. Also, we look closely at the sensitivity of *lisat* and $\text{CP}_{\text{PP}}^{fn}$ when increasing the duration of jobs in Figure 5.2.

The CP Planners perform very well on all formulations of the HTG instances of the *blocksworld* domain. Table 5.2 reveals that every configuration of the CP planner solves the full set of HTG instances. Comparing the results on *blocksworld* to the IPC instance

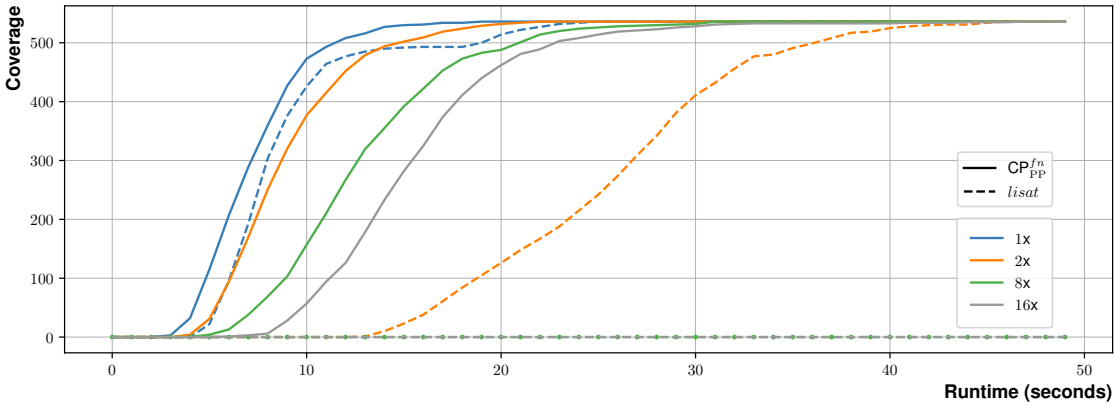


FIGURE 5.2: Ablation study on the *runtime* performance of *lisat* and CP$^{fn}_{PP}$ by scaling up the duration of the jobs in *MMPSP* by a factor of 2, 8 and 16, respectively.

set[5.3], we see the CP$^{fn}_{PP}$ planner performs strongly, too, even when the IPC instances require a much higher number of plan steps than the HTG ones.

The CP planners and baseline *lisat* planners find feasible plans for many HTG *childsnack* instances, significantly more than the state-of-the-art PL baselines, but have trouble finding optimal plans. For any given *childsnack* instance, there are many possible optimal plans that are simply permutations of each other. Without specific guidance, our planners struggle with symmetries to obtain proofs of optimality quickly. Another structural feature of optimal plans that is revealed to be problematic is the plan length. Domains where plans require many actions (HTG *ged*) are very challenging for our planners, with *lisat* performing better and the PL baseline having excellent performance in all of these instances, and *Madagascar* too when its techniques to bundle several actions apply.

In *rovers*, the *concise* encoding of dependency of preconditions on *static relations* using the *table constraints* (see Section 5.5) helps the CP$^{fn}_{PP}$ achieve better optimizing performance than the baselines. The initial states of HTG *rovers* instances may have $10,000$s of atoms to specify the *static relations*, which otherwise would make the number of constraints (5.13) and (5.10) to blow up.

The *satisficing* performance of *lisat* with *londex* constraints against $\overline{lisat}$ which does not use *londex* shows *impressive* gains in coverage by exploiting the structure of feasible plans to guide the search. The *londex* implementation of *lisat* restricts the *supporter-supportee* distance to 1, initially. If UNSAT, it increases the distance limit by 1 and solves again. It repeats the procedure until *timeout* or finds a solution. This indicates a potential for improving the *satisficing* performance of the CP encoding by designing and integrating planning-specific heuristics into the CP solvers.

Overall, all *optimizing* configurations of the CP planners perform much better than the baseline heuristic search planners on the HTG benchmark. The *functional* transformation of the PDDL tasks shows *impressive* performance gains, thereby highlighting the importance of a *concise* encoding of the planning problems. The performance of CP encoding with *simple* transformation of the PDDL task lags slightly behind *lisat*. With the functional transformation $\text{CP}_{\text{PP}}^{fn}$, the planner catches up to *lisat*, and their coverage is about the same.

However, in the *MMPSP* instances, the CP approach shows its advantage over all other baseline planners. As we can see in Figure 5.2, while the $\text{CP}_{\text{PP}}^{fn}$ is slightly ahead of *lisat* for the original problems, scaling the durations only slowly degrades $\text{CP}_{\text{PP}}^{fn}$, but immediately makes a significant difference to *lisat* since it must encode much larger time domains, while CpSat only lazily grounds the integers encoding times. We note that all the baseline heuristic planners exceeded the memory limit on the original problem set itself.

Lastly, in the IPC instances, Table 5.3, the $\text{CP}_{\text{PP}}^{fn}$ planner performs much better than the baseline *lisat*. However, all CP planners, as well as *lisat*, do not perform as well as the baseline heuristic search planners. With the exception of *blocks-3ops* and *organic-synthesis*, the coverage of the CP planners is lower than that of heuristic search planners. Heuristic search planners work well with features showcased in the IPC instances, including significantly longer plans. On the other hand, their performance suffers in problem domains with large numeric types, especially over *Resource-constrained planning(RCP)* problems [103].

5.9 Related Work

While causal encodings are the road less traveled in planning as satisfiability, there is a related work worth mentioning. Formulations based on the *event calculus* [137] exist yet are rarely acknowledged. We note that the event calculus *Initiates* predicate corresponds to our notion of output pins, *Holds* corresponds to our notion of input pin, and *Terminates* is very much equivalent to our persistence predicate in Eq. (5.9). Constraint Programming was a natural target for research seeking more compact encodings very early [149], yet our most direct inspiration was the Cpt planning system [152], which in

IPCs(opt)	Optimal Solution								
Domain	#Instances	CP_0	CP_{PP}	CP^{fn}_{PP}	LiSAT	*lmcut*	*sbd*	*cpddl*	*delfi1*
Not-grounded — action schemas do not have ground atoms									
barman-opt11	20	0	0	0	0	4	9	12	7
barman-opt14	14	0	0	0	0	0	3	6	2
blocks	35	17	19	28	26	28	30	31	27
blocks-3ops	35	19	23	35	5	26	25	30	20
childsnack-opt14	20	0	0	0	0	0	4	4	6
data-network-opt18	20	14	15	15	0*	20	17	17	17
depot	22	1	1	2	2	7	5	7	12
driverlog	20	4	4	4	5	14	12	12	15
elevators-opt08	30	2	3	5	6	20	22	23	20
elevators-opt11	20	1	1	3	4	17	18	18	16
floortile-opt11	20	0	0	0	0	6	14	14	12
floortile-opt14	20	0	0	0	0	5	20	20	17
freecell	80	6	6	6	12	15	21	22	18
ged-opt14	20	19	19	20	20	20	20	20	20
grid	5	1	1	2	1	2	2	2	3
gripper	20	1	2	2	2	7	20	20	20
hiking-opt14	20	4	4	4	8	9	14	15	19
logistics00	28	2	2	3	6	20	18	19	21
logistics98	35	0	0	1	1	6	5	5	8
miconic	150	22	23	30	32	141	104	104	136
mprime	35	31	32	33	33	22	23	25	25
mystery	30	18	18	18	19	17	13	15	17
nomystery-opt11	20	6	6	6	10	14	13	14	14
organic-synthesis-opt18	20	20	20	20	20	7	7	13	8
organic-synthesis-split-opt18	20	8	12	12	10	16	13	8	12
parking-opt11	20	1	1	1	1	2	1	1	5
parking-opt14	20	0	0	0	0	3	0	1	7
pegsol-08	30	9	8	11	20	27	28	29	28
pegsol-opt11	20	1	1	1	6	17	18	19	18
pipesworld-notankage	50	12	12	12	14	17	15	15	25
pipesworld-tankage	50	9	10	11	11	12	16	17	22
rovers	40	4	4	4	4	8	14	14	12
satellite	36	3	3	4	5	7	10	11	14
scanalyzer-08	30	10	9	13	12	9	13	13	17
scanalyzer-opt11	20	7	6	10	9	6	10	10	13
sokoban-opt08	30	0	0	0	2	24	25	28	28
sokoban-opt11	20	0	0	0	0	19	20	20	20
spider-opt18	20	0	0	0	0	6	6	6	8
storage	30	12	11	13	0*	15	14	15	17
termes-opt18	20	0	0	0	0	5	16	16	12
tetris-opt14	17	2	2	3	3	5	10	12	13
tidybot-opt11	20	1	3	3	0*	14	12	11	17
tidybot-opt14	20	0	0	0	0*	9	5	7	13
tpp	30	4	4	4	5	7	8	8	11
transport-opt08	30	6	6	6	6	12	14	14	13
transport-opt11	20	1	1	1	1	8	10	11	10
transport-opt14	20	1	1	1	2	7	9	10	9
visitall-opt11	20	8	9	11	11	10	12	12	17
visitall-opt14	20	2	3	5	5	5	6	6	13
woodworking-opt08	30	7	7	7	10	17	30	29	28
woodworking-opt11	20	2	2	2	5	12	20	20	20
zenotravel	20	7	7	8	9	13	9	0	12
Partially-grounded — action schemas have a few ground atoms									
agricola-opt18	20	0	0	0	3	0	14	12	10
airport	50	6	7	7	7	28	23	24	23
movie	30	30	30	30	0*	30	30	30	30
openstacks-opt08	30	0	1	1	2	8	30	30	30
openstacks-opt11	20	0	0	0	0	3	20	20	20
openstacks-opt14	20	0	0	0	0	0	15	16	12
parcprinter-08	30	6	6	6	0*	22	30	30	30
parcprinter-opt11	20	3	3	3	0*	16	20	20	20
pathways	30	3	4	4	0*	5	5	5	5
snake-opt18	20	3	3	6	0*	7	3	5	11
Fully-grounded — action schemas only have ground atoms									
openstacks	30	0	0	0	0	7	18	17	11
petri-net-alignment-opt18	20	0	0	0	0	3	18	20	20
psr-small	50	44	46	46	0*	49	50	50	50
trucks	30	2	2	2	0*	10	11	14	9
Total	1862	402	423	485	375	927	1090	1124	1195

TABLE 5.3: Coverage of planners on *IPCs – optimal track* benchmark domains. * indicates the domains in which the preprocessing (*parsing and encoding*) step of *lisat* rendered the instances *unsolvable*.

some of its later versions[2] used a notion of propagator for its *causal link constraints*, very close to that later formalized by Ohrimenko et al. Previous work have proposed grounded encodings that used KMS notion of *operator splitting* [40, 130], which are structurally similar to our constraints for representing ground actions. Our formulation is entirely lifted, and the only ground atoms we are forced to use are those present in initial and goal state formulas. We end acknowledging that the power of element(·) constraints and their applications to automated planning was revealed to us after the careful study of Francis et al. [53] and Francès and Geffner [51].

5.10 Conclusion

Our research demonstrates that the KMS notion of lifted causal encodings is an approach to planning as satisfiability that has become viable thanks to the notable advances in CP over the last decade. We have clearly barely scratched the surface of what is possible, as more propagator procedures follow directly from the formulation in this work, seeking powerful synergistic relations between "planning-specific" ones and general propagation algorithms. The clear limit to this approach lies in the number of variables that need to be created. We also have not really looked into the possibility of integrating or reformulating existing work that is known to enhance notably the scalability of planning as satisfiability [126].

Alternatively, and perhaps ultimately too, we need to consider PDR-like formulations [143, 39], where we only need to construct a CP model covering two plan steps. We observe that Suda's expedient of reimplementing the obligation propagation mechanism as a "Graphplan-like" algorithm strongly suggests that a CP formulation with suitably defined propagators could perform well across PPIs with diverse structures. Also, PDR formulations seem to be key for certifying unsolvability [39].

[2] Personal communication with Hector Geffner.

"An equation for me has no meaning unless it expresses a thought of God."

Srinivasa Ramanujan

6

Integrating Planning as Search and Planning as CP

Planning as search is currently the dominant approach in sequential planning. It allows the algorithms to exploit the optimal substructure of the shortest path while generating the graph incrementally. When coupled with suitable heuristics, these planners have shown state-of-the-art performance in optimizing and satisficing planning benchmarks of the International Planning Competition. On the other hand, planners based on planning as (iterated) CSP solving have recently shown state-of-the-art performance on *hard-to-ground* instances, which demonstrates the scalability of Conflict Driven Clause Learning (CDCL) in solving suitably designed SAT encodings of classical planning. Furthermore, the innovative technique of Lazy Clause Generation (LCG), which combines SAT solving and Finite Domain propagation, has enabled further generalization of CDCL with FD propagators. The resulting framework of LCG CDCL extends the usefulness of CDCL-based SAT solvers to the domain of CSP solving. In this chapter, we show how LCG

CDCL can accommodate various forward search methods with little or no modification, and possibly all of them with further research in this direction.

6.1 Introduction

CDCL [139] is an extension of the Davis-Putnam-Logemann-Loveland (DPLL) framework that allows the solver to learn a set of clauses that capture explanations of failure with the potential to reduce the search space significantly. The framework is highly adaptable, and researchers have extended it to solve a sequence of related incremental SAT problems such that the information learned is preserved in successive iterations of the CDCL loop. Moreover, Google's CpSat solver demonstrates an integration of groundbreaking Lazy Clause Generation (LCG) technology into the framework of CDCL, enabling state-of-the-art performance with a suitably designed implementation of constraints (propagators) [142]. Further generalization of the approach has been recently proposed and demonstrated in the CaDiCaL SAT solver, allowing for *external* user-propagators without modifying the internal working of the SAT-solver [44]. On the other hand, planning as search algorithms, including the suite of algorithms in the framework of width-based search, have dominated the *satisficing* planning landscape in the competitions [77].

It is crucial to recognize that the fundamental techniques driving the success of both forward search and CDCL-based approaches are not mutually exclusive. On the contrary, the framework of planning as search and that of clause learning and lazy constraint generation appear complementary. Indeed, researchers have previously combined planning-specific heuristics, based on fundamental principles related to plan properties, into the framework of CDCL in SAT solvers [123, 125]. Such integrations have been shown to consistently outperform the generic variable selection heuristics like VSIDS (Variable State Independent Decaying Sum). Furthermore, they are found to reach the same level of performance as forward search methods using heuristic guidance for many planning instances in the International Planning Competition benchmark instances [126]. However, none of these studies have explicitly compared and contrasted, or fully integrated, the two approaches: planning as search and planning as CSP solving.

This study thoroughly compares the forward search paradigm and LCG CDCL frameworks, proposing an integrated approach. For our integration, we adapt Google's

CpSat solver, which implements LCG CDCL. Our research demonstrates that blind search techniques, such as breadth-first search and depth-first search, can be seamlessly incorporated into planning as CP. This integration is achieved through adjustments to the variable selection heuristic for Lifted Sequential Planning CP encodings. Moreover, we demonstrate that value selection heuristics in CDCL solvers can be employed to choose the successor for expansion, akin to planning heuristics. Lastly, we incorporate the concept of Novelty as a constraint within CP and introduce an efficient implementation of Novelty using a polynomial-time reachability propagator. Our Lifted Encoding demonstrates enhanced performance when solved using Depth-First Novelty Search, an LCG CDCL-based algorithm incorporating forward search with novelty-based pruning.

6.2 Juxtaposing *Planning as Forward Search* and *Planning as CSP solving*

	Planning as Search	Planning as CSP solving
Search Framework	Best-First Search (BFS)[1]	CDCL, LCG CDCL[2]
Knowledge Base	Open List, Closed List, Transition Function	Constraints (Boolean, Linear), User-propagators
Heuristics	Novelty, Landmarks, Delete-relaxation	VSIDS, Activity-based, LP-heuristic
Assumptions	Novelty-Bound[3]	Plan-length[4]

Table 6.1: Examining the characteristics of Planning as Forward Search and Planning as Constraint Programming.

In Table 6.1, we present a side-by-side comparison of the approach of *planning as search* and *planning as CSP solving*, give examples of common search approaches in each, and how the knowledge base comprising of knowledge encoded in the problem description and that obtained during the search differ between the two. This is significant as the methods involved in deducing further knowledge from the knowledge bases depend on the

[1] Predominant search framework in state-of-the-art heuristic planners
[2] Framework implemented by Google's CpSat solver
[3] A novelty Bound k is assumed in width-based algorithms including IW(k) [91] and k-BFWS [92]
[4] An assumption on plan-length is made in planning as (iterated) CSP solving

languages representing the knowledge. Hence, comparing them allows us to understand the orthogonality and complementarity of the approaches. In that sense, the advantage of the CDCL framework used by CpSat lies in its representation of knowledge, SAT clauses, and constraints, which are more general when compared to the data structures of *open* and *closed* list in BFS, enabling it to capture large tracts of unreachable search space concisely. However, we must note that the value of a more expressive language of the knowledge base is limited without powerful and efficient inference procedures.

The following sections detail these approaches, explaining the procedures and discussing their strengths and weaknesses.

6.2.1 *Forward Search* Framework

Forward search is a technique for solving problems by exploring the possible states that can be reached from the given initial state until one of the goal states is reached or the search space is exhausted. It does so by incrementally generating sequences of the form $(s_0, a_1, \ldots, a_n)$ where the preconditions of all the actions in the sequence are satisfied, $\forall i \in \{1, \ldots, n\}, T(s_0, (a_1, \ldots, a_{i-1})) \models \text{Pre}_{a_i}$, and then testing whether the resulting state $s_n = T(s_0, (a_1, \ldots, a_n))$ meets the goal specification, i.e. $s_n \subseteq S_G$. Overall, the forward search strategy adheres to the following constraints.

1. The sub-sequence $(a_1, ..., a_k)$ is tested prior to $(a_1, ..., a_{k+1})$, where a_i is a ground action.
2. The sequence of actions is internally consistent, i.e., no action threatens the precondition of another.
3. All the preconditions of actions in the sequence are supported.

Thus, each sequence is chained from the initial state to the last action.

The order in which the sub-sequences are tested is further specified by the specific implementation of the forward search algorithms, and they use appropriate methods and data structures to achieve this. These algorithms have two main components: a knowledge base and a search strategy.

Knowledge Base. In forward search, the Knowledge Base contains all the necessary information to generate the next sub-sequence and perform a goal check. This includes

the transition function, which produces the successor states that can be reached from a given state by applying an action, a goal test to check whether a given action sequence results in a state that is a subset of goal states, and data structures that record the state of the search. Implementing the knowledge base, including the representation of the sub-sequences, e.g., nodes in a search tree, is implementation-specific, and the developers try to engineer a design that synergizes well. For example, the Fast Downward Planner [67] implements the successor generator using a match tree data structure, and its best-first search implementation also uses efficient data structures for maintaining open and closed lists, which represent the state of the search.

Search Strategy. The search strategy determines the next sub-sequence $(a_1, ..., a_k)$ to try under the given constraints of forward search. In other words, it is a way of choosing which node to expand next from the set of successors of the leaf nodes in the explicit search tree — a tree formed by sub-sequences that have been explored during the search in which actions of the sequence are edges, and a sequence compose a path in the tree.

The method of ordering these sub-sequences gives rise to two important classes of Forward Search: blind search and informed search. The former, blind search, uses a pre-specified ordering, e.g., lexicographic, of actions, and the choice of length of the following sequence of exploration (depth), k, makes for different variants of blind search, including breadth-first search, and depth-first search, and many others. On the other hand, the informed search uses heuristic methods that analyze the knowledge base to make a meaningful recommendation on the ordering of the sequences, generally under some assumptions of relaxation, e.g., delete relaxation, landmarks, etc. The possible arrangements of sequences in an informed search are a strict superset of the orderings that can be seen in a blind search, which means that the state of the explicit search tree, at a point in the search, can be much more complex than in a blind search. Hence, the algorithm requires more comprehensive data structures to record the state of the search.

6.2.2 *Lazy Clause Generation CDCL* Framework

Lazy Clause Generation, or LCG, is a groundbreaking technique for solving hard combinatorial problems, combining the advantages of Boolean Satisfiability (SAT) solving and Finite Domain (FD) propagation solving that forms the basis of constraint programming solvers. LCG allows us to use the expressive modeling capabilities of CP, such as global

constraints, user-defined constraints, and variables with finite domains (not limited to Boolean variables). Furthermore, it taps into the latest advancements in SAT-solving technology, such as watched literals, conflict analysis, and learning. Boolean Satisfiability solvers equip us with efficient data structures and procedures to generate effective no-goods to record failures during the search. The inference procedure of the SAT solvers, unit propagation, allows us to infer new information from the learned no-goods and the original CNF formula that prevents the search from exploring redundant portions of the search space. Moreover, the search engine of the LCG solvers can use heuristics of both solving technologies, including activity-based variable selection like VSIDS and restarts. The result is a powerful hybrid solver that can solve many more problems than SAT or FD solvers alone. Advancements in symmetry detection and breaking seek to improve this advantage further.

Google's CpSat solver implements lazy-clause generation within the framework of CDCL, more precisely, it extends the CDCL framework to account for propagators that implement the CP constraints whose interpretation as well as efficient implementations are well-defined. While CpSat does not provide an API to add user-propagators, it does define an internal interface for the propagators in its object-oriented design, which makes adding a new propagator cleaner as developers must stick to the interface. The Algorithm 1 describes the outline of Lazy Clause Generation CDCL.

The central principle behind LCG CDCL is to treat the FD propagators — inference procedures — that implement the CP constraints as lazy clause generators. Each propagator is associated with clauses representing the underlying relation, which are not explicitly specified. That is, the initial CNF formula does not contain them. Instead, the inference procedures tied to the constraint propagate when the trail of assignments updates such that the domain of the set of variables in the constraint changes and the change meets its trigger condition. In doing so, they restrict the possible values that a set of variables can take based on the relation enforced by the constraint. Upon reaching the failure condition, the conflict clauses are added to the pure SAT part of the CSP. This is made possible by efficiently modeling the finite set of choices for the integer variables in the CSP as Boolean variables, enabling the propagator to specify and propagate new conditions on integer variables as clauses. For example, we can model a variable x with finite domain $Dom_x := \{1, 11\}$ with two Boolean variables such that $x = 1$ iff b_1 is true, and $x = 11$ iff b_2 is true. Thus, the clausal description of the problem grows during the

Algorithm 1: Lazy Clause Generation CDCL

Input: CSP, decision-heuristic
Output: sat(CSP) /* sat(CSP) = true iff the CSP is satisfiable */

```
1  Procedure CDCL(CSP, decision-heuristic):
2      φ ← PureSat(CSP)
3      C ← CpConstraints(CSP)
4      level ← 0
5      ν ← {}                          /* Initialize the set of unit clauses */
6      decisions ← {}
7      while true do
8          flag-conflict ← false
9          do
10             ν' ← ν
11             foreach propagate ∈ {unit-propagate} ∪ Propagators(C) do
12                 flag-conflict, ν̂, c ← propagate(φ, ν)
13                 if flag-conflict = true then
14                     if level = 0 then
15                         return false
16                     else
17                         φ ← φ ∧ c
                           /* backjump to a previous level such that c holds  */
18                         ν ← undo-assignments(c, ν)
19                         level ← compute-level(ν, decisions)
20                         flag-conflict ← true
21                         break
22                     end
23                 else
24                     ν ← ν ∪ ν̂
25                 end
26             end
27         until ν' = ν or flag-conflict
28
29         l ← decision-heuristic(φ, ν)          /* such that {l} ∩ ν = ∅ */
30         if l = nill then
               /* all propositional variables are assigned                */
31             return true
32         else
33             level ← level + 1
34             ν ← ν ∪ {l}
35             decisions ← decisions ∪ {(level, l)}
36         end
37     end
```

search, until a fixed point is reached where all the Boolean variables are assigned and all the constraints are satisfied, line 31 in the Algorithm.

While the framework of clause generation isn't set in stone, the propagator implementations should at least detect any conflicts — a contradiction or an inconsistency in the variable assignment, and generate a clause that explains the cause of the failure, adding it to the CNF formula. FD propagators can be designed to propagate more aggressively and eagerly add clauses to reduce the search space. A good example of lazy clause generation is the propagator for the all-different constraint, which is a global constraint that ensures that no two variables in the specified set are assigned the same values. The all-different constraint can be implemented as a maximum bipartite matching problem [74], which checks whether the constraint is feasible by finding a maximum cardinality matching. If the matching does not cover all the variables in the set, then a conflict clause is generated and added to the CNF. Additional inference procedures can be employed to eagerly reduce the domain of the variables, including domain consistency checks and bound consistency checks [74] that can be propagated as unit clauses specifying disallowed assignments.

LCG CDCL has its challenges and limitations, and although the approach to lazily generate clauses reduces the computational burden on the SAT solver, it may also prevent the solver from benefitting from combined inference over the whole set of clauses made implicit through FD propagators. Furthermore, it is possible that for specific constraint satisfaction problems, the propagators need to incrementally generate clauses of the same order of magnitude (in count) as the set of clauses represented by the propagator, in which case the lazy clause generation loses its advantage. Hence, the question of how to design and implement efficient clause generators is quite essential and challenging. Some constraints for which efficient inference procedures are already known are more accessible to implement as lazy propagators, such as linear arithmetic and cardinality constraints. Designing efficient inference propagators for user-defined constraints is more complex and sometimes intractable. Another concern is balancing the trade-off between eagerly adding clauses and navigating a more extensive search space. Eagerly generating clauses can reduce the search space due to inference methods employed by the SAT solver. Still, it increases the overhead associated with more significant memory usage and slows down the clause manager. On the other hand, lazily adding clauses may have

the opposite effect. All these concerns must be carefully considered and addressed to allow the components and propagators to synergize well in the framework of LCG CDCL.

6.3 Planning as Forward Search LCG CDCL

In this section, we explain an approach to emulate forward search in the framework of planning as (iterated) CSP solving by simply modifying the CP search heuristics. The idea is not unprecedented; Rintanen has previously shown simulations of IDA* with SAT search [123].

Typically, a CP search heuristic consists of two components: a **Variable Selection Heuristic** and a **Value Selection Heuristic**. Variable selection heuristics decide the order in which the variables are selected for branching, and the value selection heuristics fix the value of a selected variable. There are two popular approaches to designing such heuristics: autonomous search heuristics and programmed search heuristics. Autonomous decision strategies alter their behavior as the search progresses using some method and are better suited to complex systems of constraints where the solution structure is not apparent. Examples of autonomous heuristics include activity-based search heuristics (VSIDS) and impact-based search strategy. On the other hand, programmed search is used to exploit human knowledge of the structure of the solution set. In designing forward-search in CP, we substitute the autonomous search, employed by CpSat, with programmed search strategies.

We view the forward search strategy in planning as a programmed variable selection heuristics, in which we branch upon the CP model variables in the forward direction, i.e., the assignment to variables representing the sequence $(a_1, \cdots, a_k)$ occurs before $(a_1, \cdots, a_{k+1})$. We note that in our CP encoding of Lifted Sequential Planning, described in Section 5.4, the sequence $(a_1, \cdots, a_k)$ is represented by values assigned to the variables $(act_1(\overline{arg}_1) \cdots, act_n(\overline{arg}_n))$. Hence, we must branch upon the action variables in the forward direction, from slot 1 to slot N_π to ensure the constraints discussed in Section 6.2.1 are met, that is:

1. A variable of slot $j+1$ is selected if and only if the *necessary* variables of the slot 1 to j have been branched on and assigned fixed values.

2. Fixing the variables act_j and in_{jk}[5]are *necessary* to reveal threats to the preconditions in_{jk}.

3. The variables spt_{jk} are *necessary*, guaranteeing that all preconditions are supported.

Forward search planners generally maintain an explicit representation of the state, which allows them to use efficient methods to ensure that the preconditions of the suffix a_{k+1} are supported and without any threats, e.g., match-tree to compute the set of applicable actions. On the other hand, in planning with CP, we depend on the general inference methods and propagators to remove inapplicable actions from the domain variables $act_{k+1}(\overline{arg}_{k+1})$. Hence, we are required to branch on and fix the *necessary* variables that reveal potential inconsistencies — unsupported preconditions and threats to supported preconditions to prevent any inconsistency in the prefix.

Next, we explain our approach to implement the paradigms of Depth-First Search and Breadth First Search as Depth-first LCG CDCL and Breadth-First LCG CDCL, respectively, in a manner that is consistent with the forward-search requirements discussed in this section.

6.3.1 Depth-First Search in LCG CDCL

Depth-first search is a Blind search algorithm that does not depend on heuristic guidance. The Algorithm not only tries the prefixes $(s_0, a_1, \ldots, a_k)$ before $(s_0, a_1, \ldots, a_{k+1})$ in the forward direction but also does so sequentially until it finds a sequence that maps into a goal or a dead-end is reached, equivalently $\mathcal{T}(s, \langle a_1, a_2, \ldots, a_k\rangle) \subseteq S_G$ or $\nexists a_{k+1}$, $(\mathcal{T}(s, \langle a_1, a_2, \ldots, a_k\rangle), a_{k+1}) \in Dom_T$. If a path to a goal is found, we have found an upper bound; otherwise, a dead-end is encountered, and the search backtracks to the previous level. In this section, we explain a programmed search strategy that implements the notion of Depth-first Forward Search by ensuring that the CP variables are assigned to that effect.

In our CP encoding, the prefix $(s_0, a_0, \ldots, a_k)$ is captured by the assignment to CP variables $(\mathcal{V}(in_0), \mathcal{V}(act_1)(\mathcal{V}(\overline{arg}_1)), \ldots, \mathcal{V}(act_k)(\mathcal{V}(\overline{arg}_k))$, where $\mathcal{V} : \mathcal{X} \mapsto \mathbb{Z}$ is partial function that encodes variable assignments during the CP search, $\mathcal{V}$ is *total* and defined

[5]Data of output pins out_{jk} is fixed based on the the assignments to in_{jk} and act_j.

Algorithm 2: Depth-first(Blind-*lexicographic value selection*) Search Heuristic

Input: $\overline{act}$, Arg, $[Act]$, $[U]$
/* $\overline{act}$ is a vector of size N_π, and Arg is a matrix of size $N_\pi \times K_\alpha$, representing action-schema and argument variables, respectively, in the Lifted CP Encoding, Table 5.1. */
Output: decision
/* A literal implying variable-value assignment */
1 $d \leftarrow 0$ /* length of the prefix $(s_0, a_1, \ldots, a_d)$ */
2 $\texttt{action} \leftarrow ((0)_{\times K_\alpha+1})$ /* zero-vector of size $K_\alpha + 1$ */
3 $\texttt{row} \leftarrow ((0)_{\times N_\pi})$ /* row-indexes of the matrix $lex([Act] \times [U]^{K_\alpha})$ */
/* representing action sequence $(a_1, \ldots, a_{N_\pi})$ */
4 $\texttt{col} \leftarrow 0$ /* column-index in the vector $lex([Act] \times [U]^{K_\alpha})_{\texttt{row}_d}$ */
5
6 **Procedure** dfs-decision():
7 **if** $\texttt{col} < K_\alpha$ /* *There exists* a partially applied action choice */
8 **then**
9 $\texttt{col} \leftarrow \texttt{col} + 1$
10 **return** $(arg_{d(\texttt{col}-1)} = \texttt{action}_{d,\texttt{col}})$
11 **else**
12 $\texttt{col} \leftarrow 0$
13 $d \leftarrow d + 1$
14 $\texttt{row}_d \leftarrow \texttt{row}_d + 1$
15 $\texttt{action}_d \leftarrow lex([Act] \times [U]^{K_\alpha})_{\texttt{row}_d}$
16 **return** $(act_d = \texttt{action}_{d,\texttt{col}})$
17 **end**

for all $\mathcal{X}$ when a solution has been found. Hence, our programmed search heuristics must ensure that the variables act_i and arg_{ij} are fixed sequentially in the forward direction. Algorithm 2 does precisely that in CpSat. Given the variables act_i and arg_{ij}, and the domain of action schema Act and universe of objects O, the Algorithm returns a *literal* associated with $(x = \mathcal{V}(x))$, where $x \in \mathcal{X}$ and $\mathcal{V}(x)$ are a CP variable and its value, respectively, selected by the Depth-first search decision heuristic. The *literal* is then added to the set of unit clauses in the LCG CDCL, followed by recursive calls to the propagators until a fixed point is reached. The decision heuristic increments the length of the prefix once the literals associated with sequence $\mathcal{V}(act_k)(\mathcal{V}(\overline{arg}_d))$ are set to *true*, exploring the search tree in the forward direction. It does so by selecting the next action at depth $d + 1$ from the lexicographic ordering of the set $[Act] \times [U]^{K_\alpha}$. Whenever a dead-end is reached, the CDCL algorithm backtracks the state of the search to the previous level(of the CP search), which is exactly the expected behavior of the Depth-first forward search in planning. We encode the variables $d, \texttt{action}, \texttt{row}, \texttt{col}$ in the state of the CP search, which allows them to be reversible, i.e., revert to the previous

Algorithm 3: Breadth-first(Blind-*lexicographic value selection*) Search Heuristic

Input: $\overline{act}$, Arg, $[Act]$, $[U]$
Output: `decision`

```
   /* line 1 to 4 from Algorithm 2                                    */
1  ...
2  max-row ← num-rows(lex([Act] × [U]^{K_α}))
3  Procedure bfs-decision():
       /* line 7 to 10 from Algorithm 2                               */
4      ...
5      else if row < max-row then
6          backtrack(d-1)
7          return bfs-decision()
8
       /* line 11 to 16 from Algorithm 2                              */
9      ...
```

assignment upon backtrack. Thus, we achieve Depth-first forward search in CpSat with programmed search heuristics without additional adjustments.

We can also adjust the depth-first forward search heuristic to explore the search tree in the Breadth-first order of exploration by setting a depth limit and performing an iterative deepening search. However, this requires us to implement an additional `backtrack` procedure, which allows the heuristic to fix a depth limit. As described in Algorithm 3, the heuristic backtracks the state of the CP search whenever the depth limit is reached. In LCG CDCL, backtracking the search is a standard procedure, which can be used as building blocks to implement the `backtrack` method.

6.4 Depth-First Novelty Search

Width-based algorithms using the general notion of state novelty have been shown to have state-of-the-art performance in the classical planning benchmarks. Moreover, we can apply these algorithms to model-free problems where the transition system is encoded as a black box simulator [6]. Novelty-based planning algorithms, including Iterated-Width, which involves sequential calls to novelty-bounded IW(k), and *polynomial* Best-First Width search, k-BFWS, [91, 92] use the *Novelty bound* to prune the nodes in the search tree, assuming that the goal is reachable from the nodes of novelty value lower than this *bound*. These search algorithms are especially well-suited for problems where the lower bound on the maximum state novelty of states in solution plans is small, as these

algorithms are *polynomial* on this lower bound on the novelty measure. The state novelty measure has also been used as a heuristic measure to guide exploration in BFWS [93] in tandem with heuristics that exploit distance to the goal, e.g., goal counting, enabling the algorithm to achieve state-of-the-art performance.

This section defines the concept of *Novelty bound* in width-based algorithms as a CP constraint. Then, we present an efficient procedure for the user-propagator that implements it. The proposed algorithm *Depth-first Novelty search* applies the novelty constraint along with the depth-first search strategy. Furthermore, the prefixes $(s_0, a_1, \ldots, a_k)$ mapping into states with lower Novelty are prioritized.

6.4.1 Defining *Novelty Bound* in Planning as a *CP Constraint*

As per the vanilla definition of novelty [91], the Novelty $w(s)$ of the newly generated state s is the size of the smallest *new* tuple in the set of all possible tuples (conjunctions) of atoms in s. A tuple t is considered *new* if none of the states generated before s model the tuple t, i.e. $\forall s' \in \mathcal{P}(s), s' \not\models t$, where we denote the set of states generated before s as $\mathcal{P}(s) = \{\ s' \,|\, e(s') < e(s)\}$, and the mapping $e : S \mapsto \mathbb{N}$ ranks a state in the order of expansion. Equation 6.1 presents a precise definition of the novelty measure.

$$w(s) := \min\{card(t) \mid s \models t, \nexists s' \in S, e(s') < e(s) \wedge s' \models t\} \tag{6.1}$$

A keen eye would notice that the definition of Novelty depends on the expansion order of the states. Hence, the state novelty measure has an implicit parameter e, the expansion order determined by the search strategy, apart from the evaluated state. In blind search algorithms, the expansion order is pre-determined, e.g., lexicographic. On the other hand, in informed/heuristic search algorithms, the expansion order is implicit and revealed as the search progresses.

Width-based search algorithms with Novelty bound k, e.g., IW (k), require that the highest novelty value of a state in an admissible path is bounded by k. That is, $w(s_i) \leqslant k$ in the sequence $(s_0, \ldots, s_i, \ldots, s_n)$ induced by a solution plan $\pi = (a_1, \ldots, a_n)$, where $s_i = \mathcal{T}(s_0, \langle a_1, \ldots a_i \rangle), i > 0$. Next, we explain how we define this constraint for our CP encoding of Lifted Sequential Planning.

***Novelty bound* as a CP constraint**

Our CP encoding only implicitly encodes the states in the sequence $(s_0, a_1, s_1, \ldots, s_g)$. Hence, we generalize the definition of novelty measure to prefix $(s_0, a_1, \ldots, a_i)$, which is explicitly represented by in_{0j}, act_i and arg_{ij} CP model variables.

$$w(\pi) = \min\{Card(t) \mid \mathcal{T}(s_0, \pi) \vDash t, \nexists \pi', e(\pi') < e(\pi) \wedge \mathcal{T}(s_0, \pi') \vDash t\} \tag{6.2}$$

where $\pi = (a_1, \ldots, a_k)$ is an admissible sequence of actions that meets the Constraints 2.2, and the expansion order e is defined on the set of all admissible sequences.

In the Iterated-width(IW) search that uses the breadth-first search strategy, the expansion order could be defined lexicographically[6].

$$\begin{aligned} e_{BrFS}(\langle a_1, \ldots a_i \rangle) < e_{BrFS}(\langle s_0, a'_1, \ldots a'_j \rangle) \\ \iff Lex(\langle a_1, \ldots, a_i \rangle) < Lex(\langle a'_1, \ldots, a'_j \rangle) \end{aligned} \tag{6.3}$$

Similarly, for Depth-first search.

$$\begin{aligned} e_{DFS}(\langle a_1, \ldots a_i \rangle) < e_{DFS}(\langle a'_1, \ldots a'_j \rangle) \\ \iff Lex(\langle a_1, \ldots, a_k \rangle) < Lex(\langle a'_1, \ldots, a'_k \rangle), \\ \vee\ (Lex(\langle a_1, \ldots, a_k \rangle) = Lex(\langle a'_1, \ldots, a'_k \rangle) \wedge i < j), k = \min\{i, j\} \end{aligned} \tag{6.4}$$

We use the definition of novelty measure in Eq. 6.2 to define the *novelty bound* as a CP Constraint, where $e \in \{e_{BrFS}, e_{DFS}\}$.

Definition 6.1. The **Novelty bound** on the solution sequence to the CP Encoding of Sequential Classical Planning is defined as a constraint that enforces an upper bound on the novelty value of every subsequence of the plan $\pi = (a_1, ..., a_i)$, Eq. 6.1.

$$\forall i \in \{1, ..., N_\pi\}, w(\langle a_1, ..., a_i \rangle) \leqslant \bar{\omega} \tag{6.5}$$

[6] *Lex* rank is defined by the lexicographic ordering of sequences. Order between two sequences is decided by the first non-coincident pair (a_k, a'_k) for equal-length sequences, otherwise ordered by the length.

***Extended* definition of Novelty *given partition functions*.** In implementing the *Depth-first novelty search*, we used an extended definition of Novelty, shown to have an empirical advantage [92, 93].

Definition 6.2. The **novelty** *given partition functions* H, $w(\pi) = w_{\langle H \rangle}(\pi)$, of an admissible sequence $\pi = (a_1, \ldots, a_n)$ is k, iff (1) exists a tuple[7] t of *minimum* size k, s.t. $\mathcal{T}(s_0, \pi) \models t$, (2) $\forall \pi' \in \mathcal{P}(\pi, H)$, $\mathcal{T}(s_0, \pi') \not\models t$.

$$w(\pi) = \min\{Card(t) \mid \mathcal{T}(s_0, \pi) \models t, \nexists \pi' \in \mathcal{P}(\pi, H), \mathcal{T}(s_0, \pi') \models t\} \tag{6.6}$$

where $\mathcal{P}(\pi, H) = \{\ pi' \mid \pi',\ \forall h \in H, h(\pi') = h(\pi) \wedge e(\pi') < e(\pi)\}$ is the set of admissible sequences of action, partitioned by the evaluation of functions in H.

6.4.2 Novelty Constraint as a polynomial reachability propagator

We implement the constraint of *Novelty bound* defined in Eq. 6.1 as a *user-propagator* in the framework of LCG CDCL, following the extended definition of the novelty measure, Definition 6.2. For this, we assume that the admissible sequences are explored in lexicographic order, e_{DFS}, defined in Eq. 6.4, which allows us to implement the constraint with an efficient propagator which we describe in Algorithm 5. The propagator implementation exploits the fact that the lexicographic order of exploration of admissible sequences, e_{DFS}, holds when we use the Depth-first search heuristic, which allows us to do away with the explicit computation of the set $\mathcal{P}(\pi, H)$ for novelty computation. Moreover, we approximate the novelty measure using the method described in Section 4.3, which gives us strong complexity guarantees for the computation of state novelty — *linear* time and consuming a fixed amount of memory. Now, we explain the inner workings of the Novelty propagator.

In implementing novelty propagator in CpSat, we introduce an auxiliary variable var_w that records the current Novelty bound. The propagator fixes the var_w to the Novelty bound $\bar{w}$ at the beginning of the CP search. This step is especially vital when we wish to explore the plan prefixes in the order of their Novelty, for which we sequentially increase the Novelty bound $\bar{\omega}$ from 1 to an upper bound k. Fixing the var_w to the Novelty bound as an assumption at the current iteration ensures that the learned clauses are still valid when the search restarts with a higher novelty bound, allowing us to avoid reinitializing

[7] Conjunction of atoms.

Algorithm 4: Novelty Computation

Parameter: $\mathcal{T} : S \times \Pi \mapsto S$, the transition function
Parameter: $H = \{h_g\}$, goal count used for partitioning
Parameter: $\mathcal{N} = |\text{Co}_{h_g}|$, number of partitions

```
novely-table ← (BloomFilter()_×N)
Procedure w(π):
    Output: Novelty value of π
    s ← T(s_0, π)
    p ← h_g(s)
    k ← ∞
    tuple-sample ← random sample of tuples of s
    foreach t in tuple-sample do
        if t exists in novely-table_p then
            novely-table.add(t)
            k ← min{k, |t|}
        end
    end
return k
```

Algorithm 5: Implementation of the Novelty Propagator

Parameter: act_i and arg_{ij}, CP variables that represent an action sequence
Parameter: $\bar{\omega}$, the novelty bound
Parameter: $\mathcal{V}$, mapping that captures the assignment at any point in the search

```
Procedure novely-bound-propatator():
    Output: conflict-clause, unit clauses
    if V is undefined for all act_i and arg_ij variables then
        return (nil, {var_w) = ω̄})
                /* Auxiliary variable var_w captures the novelty bound */
    end
    d ← current depth of the depth-first forward search
    if V is defined for all act_i and arg_ij variables, i ⩽ d then
        /* V maps variables to values, d is the length of the prefix  */
        π ← (V(act_1)(V(arg_1)), ..., V(act_d)(V(arg_d)))
        if w(π) > ω̄ then
            c ← ¬ ( ⋀_{i=1}^{d} (act_i = V(act_1)) ∧ ∧_{j=1}^{K_α} (arg_ij = V(arg_ij)) ∧ (var_w > ω̄)) )
            return (c, {})
        end
    end
return (nil, {})
```

the CpSat solver and preserve the learned clauses and the state of activity-based SAT heuristic.

Whenever we encounter a plan prefix π whose novelty value exceeds the Novelty bound, $\bar{\omega}$, we add a conflict-clause that is a conjunction of *literal* associated with assignment to the act_i and arg_{ij} action variables, that encode the plan prefix π, and var_w, enabling us

to capture the fact that the novelty value of π exceeds $\bar{\omega}$. The LCG CDCL algorithm automatically backtracks to the previous decision level upon observing the conflict, and the depth-first decision heuristics presents it with the choice plan prefix to try next.

6.5 Empirical Results

We experimented with our algorithms by extending Google's CpSat solver, integrating the Depth-first search strategy and novelty propagator into the CDCL loop, as described in Algorithm 1. We tested the two implementations of forward search strategies: *depth-first search (blind)* and *depth-first novelty search*. We will refer to them as DFS^{fn}_{PP} and DFNS^{fn}_{PP}, respectively. We compare them against the CP^{fn}_{PP} planner, which solves the model of CP with our implementation of *persistence proopagator*, explained in Chapter 5. All planners are tested on the HTG benchmarks and the problems from the *optimal track* of the IPC [77].

Implementations of Planners. The DFS^{fn}_{PP} is same as CP^{fn}_{PP}, except that it follows the *blind* search strategy of depth-first forward search, Algorithm 2, instead of the *autonomous* activity-based heuristics. On the other hand, we configure DFNS^{fn}_{PP} to run depth-first novelty search (DFNS) with a novelty bound of 1. DFNS is incomplete, and many IPC instances, barring *visitall*, cannot be solved within bound 1. However, our main aim here is to use novelty search as a guidance measure to train the activities of the variables for activity-based heuristics. That is, the DFNS^{fn}_{PP} planner hands over the control to activity-based heuristics after the DFNS terminates without finding a solution, carrying over the activity weights of the variables. For the small Novelty bound of 1, the algorithm terminates quickly, whether or not it finds a solution, making it suitable as a quick pre-processing step.

Performance over benchmarks. From Table 6.2, we gather that the $\text{DFNS}^{fn}_{\text{PP}}$ planner performs better than its autonomous heuristic counterpart, $\text{CP}^{fn}_{\text{PP}}$ in the IPC instances, solving 36 additional instances, improving the coverage to 521 from 485. There are multiple reasons for the performance gain. Firstly, we highlight the *visitall* domain, where we solve 10 more problems than $\text{CP}^{fn}_{\text{PP}}$ out of the 40 total. Here, many instances are solved within the Novelty bound of 1. The planner also sees improvements in coverage of *Blocks*, *freecell*, and *pegsol* domains, which are not directly related to the novelty search; instead, novelty search contributes to training the weights of activity-based heuristic

IPCs(opt)	Optimal Solution			
Domain	#Instances	DFS^{fn}_{PP}	$DFNS^{fn}_{PP}$	CP^{fn}_{PP}
Not-grounded — action schemas do not have ground atoms				
barman-opt11-strips	20	0	0	0
barman-opt14-strips	14	0	0	0
blocks	35	16	35	28
blocks-3ops	35	17	35	35
childsnack-opt14-strips	20	0	0	0
data-network-opt18-strips	20	11	15	15
depot	22	1	2	2
driverlog	20	3	4	4
elevators-opt08-strips	30	1	5	5
elevators-opt11-strips	20	0	3	3
floortile-opt11-strips	20	0	0	0
floortile-opt14-strips	20	0	0	0
freecell	80	6	12	6
ged-opt14-strips	20	19	20	20
grid	5	1	1	2
gripper	20	1	2	2
hiking-opt14-strips	20	2	4	4
logistics00	28	1	3	3
logistics98	35	0	1	1
miconic	150	26	30	30
mprime	35	24	33	33
mystery	30	16	18	18
nomystery-opt11-strips	20	6	6	6
organic-synthesis-opt18-strips	20	18	20	20
organic-synthesis-split-opt18-strips	20	9	10	12
parking-opt11-strips	20	0	1	1
parking-opt14-strips	20	0	0	0
pegsol-08-strips	30	8	18	11
pegsol-opt11-strips	20	1	3	1
pipesworld-notankage	50	8	12	12
pipesworld-tankage	50	6	11	11
rovers	40	4	4	4
satellite	36	3	4	4
scanalyzer-08-strips	30	11	13	13
scanalyzer-opt11-strips	20	8	10	10
sokoban-opt08-strips	30	0	1	0
sokoban-opt11-strips	20	0	0	0
spider-opt18-strips	20	0	0	0
storage	30	11	13	13
termes-opt18-strips	20	0	0	0
tetris-opt14-strips	17	2	3	3
tidybot-opt11-strips	20	1	3	3
tidybot-opt14-strips	20	0	0	0
tpp	30	4	5	4
transport-opt08-strips	30	5	6	6
transport-opt11-strips	20	0	1	1
transport-opt14-strips	20	1	1	1
visitall-opt11-strips	20	9	16	11
visitall-opt14-strips	20	3	10	5
woodworking-opt08-strips	30	5	8	7
woodworking-opt11-strips	20	0	3	2
zenotravel	20	7	8	8
Partially-grounded — action schemas have a few ground atoms				
agricola-opt18-strips	20	0	0	0
airport	50	7	7	7
movie	30	30	30	30
openstacks-opt08-strips	30	0	1	1
openstacks-opt11-strips	20	0	0	0
openstacks-opt14-strips	20	0	0	0
parcprinter-08-strips	30	5	6	6
parcprinter-opt11-strips	20	2	3	3
pathways	30	1	4	4
snake-opt18-strips	20	2	8	6
Fully-grounded — action schemas only have ground atoms				
openstacks-strips	30	0	1	0
petri-net-alignment-opt18-strips	20	0	0	0
psr-small	50	29	46	46
trucks-strips	30	1	2	2
Total	1862	352	521	485

TABLE 6.2: Coverage of planners on *IPCs – optimal track* benchmark domains. * indicates the domains in which the preprocessing (*parsing and encoding*) step of *lisat* rendered the instances *unsolvable*.

Hard-to-ground	Optimal Solution			Satisficing Solution		
Domain	$\text{DFS}^{fn}_{\text{PP}}$	$\text{DFNS}^{fn}_{\text{PP}}$	$\text{CP}^{fn}_{\text{PP}}$	$\text{DFS}^{fn}_{\text{PP}}$	$\text{DFNS}^{fn}_{\text{PP}}$	$\text{CP}^{fn}_{\text{PP}}$
blocks-3ops(40)	36	40	40	34	40	40
blocks-4ops(40)	40	40	40	40	40	40
childsnack(144)	0	48	48	4	144	144
ged(156)	22	34	31	22	39	29
ged-split(156)	22	26	24	19	47	38
logistics(40)	15	35	36	15	24	40
org-syn-MIT(18)	17	16	18	17	18	18
org-syn-alk(18)	18	18	18	18	18	18
org-syn-orig(20)	8	13	15	4	14	9
pipes-tkg(50)	11	17	17	10	34	25
rovers(40)	2	8	8	1	9	10
visitall3D(60)	34	35	34	29	49	49
visitall4D(60)	30	33	35	26	44	44
visitall5D(60)	38	31	33	25	43	44
Total	293	394	397	264	563	548

TABLE 6.3: Coverage of planners on *hard-to-ground* benchmark domains.

search, which is then able to solve more problem optimally. However, this notion does not seem to work every time. As we observe in Table 6.3, the coverage over *hard-to-ground* instances reduces slightly, indicating that the weights learned have some element of luck involved.

On the other hand, the coverage of $\text{DFS}^{fn}_{\text{PP}}$ is impressive in both IPC and HTG, given that it involves a straightforward complete *blind* depth-first search without any heuristic guidance, and solving this many instances highlights the power of clause learning combined with unit propagation. The planner solves 352 and 293 instances in IPC and HTG instances, respectively.

6.6 Conclusion

This work demonstrates that integrating methods from planning as a search paradigm is viable by customizing LCG CDCL heuristics and introducing planning-specific propagators in the LCG CDCL loop. We formalize the notion of Novelty bound from width-based planning as a CP constraint and showcase that programmed heuristics that emulate depth-first search in planning and propagator implementation of Novelty bound can help improve the planner's performance while maintaining the advantages of clause-learning in CDCL. Yet, we have barely scratched the surface here. Potentially, informed search

heuristics, like best-first search and hill-climbing, could be integrated into the LCG CDCL framework with suitably defined, planning-specific, heuristics and propagators.

"You could never convince a monkey to give you a banana by promising him limitless bananas after death in monkey heaven."

Yuval Noah Harari

7

CP encodings for Temporal Planning with *Required Concurrency*

Temporal Planning problems add the concern of concurrency on top of assignment, routing, and sequencing, which exist in sequential planning. However, most temporal planners do not fully address the concern of concurrency. Indeed, many focus on a smaller set of simple temporal planning instances, where given the partial ordering of actions in the temporal plan or schedule, a total order can always be obtained using a straightforward topological sort. Within this viewpoint, the temporal planning problems can be classified into *simple* and *required concurrency* instances. Unlike *simple* problem that can be reduced to a STRIPS problem and solved using sequential planning algorithms, instances with *required concurrency* necessitate concurrent (overlapping) execution of actions, where the structural dependencies between actions in the set of plans drive this need for concurrency. In this work, we address the challenge posed by problems with *required concurrency*, taking direct inspiration from the CPT planner's POCL formulation of temporal planning and extending CPT's CP encoding to handle temporally expressive

planning languages. We are happy to report that the resulting planner performs very well on the instances with required concurrency requirements in *optimal* planning. However, the approach performs poorly in problems requiring long, totally ordered (sequential) execution of actions in plans.

7.1 Introduction

Most early works in temporal planning addressed temporally simple problems, repurposing classical planning approaches for temporal planning [59, 37, 42]. However, this approach cannot handle the general class of temporal planning, as many problems *require* actions to execute concurrently that go beyond the expressivity of temporally simple languages. Also, while minimizing the total cost of actions is the primary objective in optimal classical planning, we include makespan optimality as an essential new criterion to determine plan quality in temporal planning. Overall, the two problems are, in essence, different, requiring different computational approaches to solve. Many works have attempted to address this issue over the past two decades, including the noteworthy work on forward-chaining planners starting with CRIKEY [64, 27]. The subsequent works on POPF [25] and OPTIC [8] are asserted to be *complete* for temporally expressive languages. However, these planners are unable to scale up on small instances of *Cushing* domain from IPC 2018, highlighting a need for new approaches. In this work, we attempt to address the issue from a CP standpoint, using the *partial order causal link* (POCL) formulation of temporal planning.

The primary appeal of *partial order causal link* (POCL) planning [98] for temporal planning is its expressive power. It contains more expressive search nodes that capture local constraints over a set of steps (actions), compared to a search node in planning as search that only consists of true atoms at a single step of the progression and regression search. Many researchers have shown interest in adopting partial-order plans in the temporal context, including the current state-of-the-art optimizing planner, OPTIC [8], which implements the forward-chaining search strategy for partial-order planning. Another work on the constraint-based approach to Temporal POCL planning, CPT [152], has inspired our research. However, CPT is unable to handle problems with *required concurrency* [31]. In our work, we extend CPT's POCL formulation of the temporal planning problem to accommodate instances with required concurrency. We develop CP encodings of this

extended POCL formulation, extending the POCL encoding of CPT, tapping into the recent advancements in CP technology for *makespan* optimal planning [49]. We implement the encoding in the CpSat [113] solver, which implements the Lazy Clause Generation (LCG) approach to CP [108] and has shown state-of-the-art performance in scheduling benchmarks [142].

Chapter Outline. We begin by explaining the structural relationship between durative actions that results in *required concurrency* requirement. After this, we discuss POCL planning and the temporal POCL formulation used by the CPT planner. Then, we present our extended CP encoding of Temporal POCL planning that handles *required concurrency*. We implement the encoding in the CpSat solver and present an empirical analysis of the planner performance.

7.2 *Required Concurrency* in temporal planning

A set of durative actions in a temporal planning problem can have dependencies that may necessarily require them to have overlapping execution. Such problems are said to have a property of *Required Concurrency* [31]. This is explained with the help of Figure 7.1, which illustrates scenarios in which pairs of actions must execute concurrently. In the first case, two actions a_1 and a_2 must overlap in time to avoid interfering with each other's *at start* preconditions, i.e., a_1 deletes p_2 required by a_2 and a_2 deletes p_1 required by a_1. Assuming that a_1 and a_2 are the only actions in the instance, then in no circumstances (feasible plans) can the two be separated, and hence, the instance has *required concurrency*. Similarly, in the second example, the two actions achieve each other's *at end* preconditions with *at start* effects, and hence each must start before the other ends. The third case is that of *mutex* effects requiring that deletions be carried out before additions. The fourth and final case is concerned with actions that manage the life cycle of a process, for example, a kiln in our temporal machine shop example in Figure 3.1, requiring the action dependent on the process to finish before the process ends. More importantly, *required concurrency* exists in *makespan* optimal planning, where all optimal plans require one or more actions to execute concurrently.

Definition 7.1. A temporal planning instance is considered to have **Required Concurrency**, iff *all* solutions to the problem have concurrent plans.

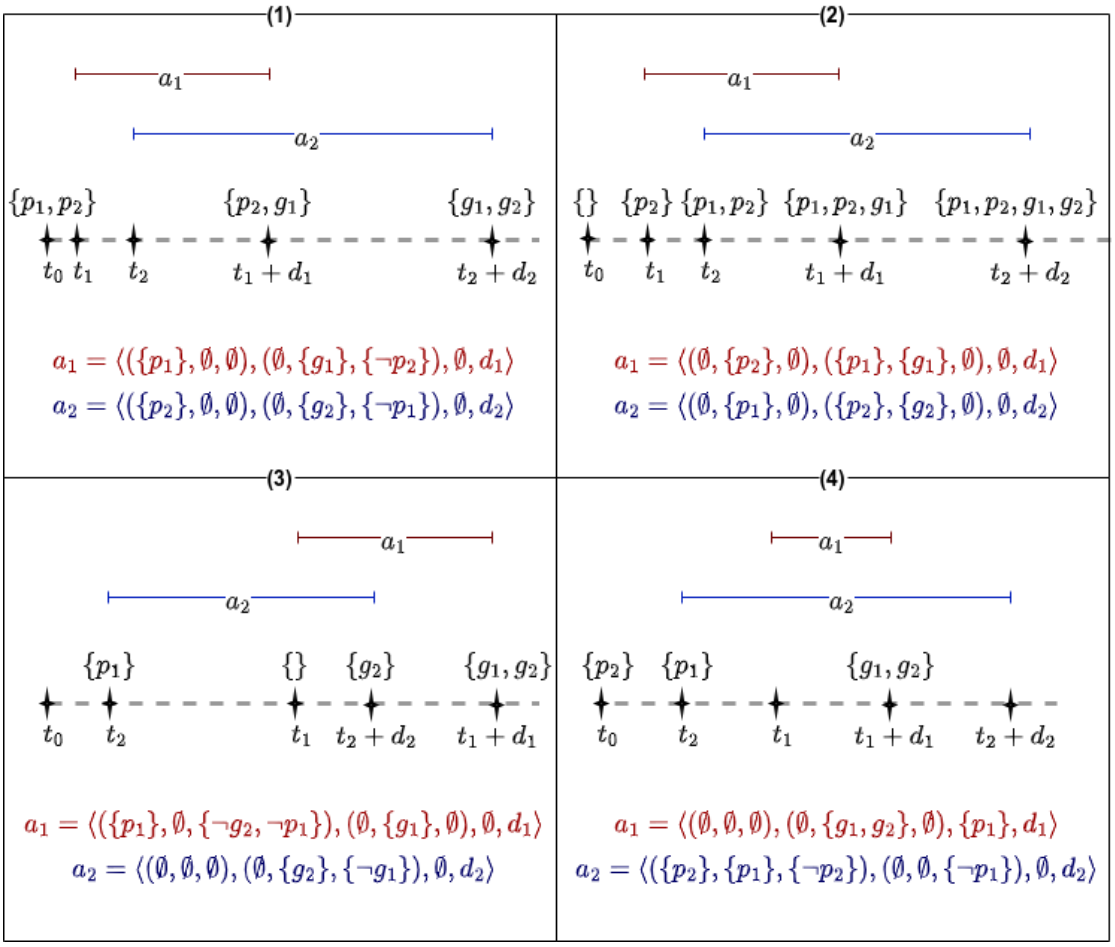


FIGURE 7.1: Illustration of *required concurrency* scenarios. Each example consists of a temporal planning instance with operator set $O := \{a_1, a_2\}$ and goal set $G := \{g_1, g_2\}$. Each operator a_i is defined as a tuple $\langle o_{\vdash}, o_{\dashv}, \text{Pre}_i^{\leftrightarrow}, d_i\rangle$, where $o_{\vdash}$ and $o_{\vdash}$ are *classical* snap actions that execute at t_i and $t_i + d_i$, respectively, and $\text{Pre}_i^{\leftrightarrow}$ are the *overall* conditions. A snap action is a tuple $o := (\text{Pre}_o, \text{Add}_o, \text{Del}_o)$.

A *plan* $\pi := \langle(o_1, t_1, d_1), (o2, t_2, d_2), \dots\rangle$ is *concurrent* iff there exists a pair (i, j) for which $t_i \leqslant t_j \leqslant t_i + d_i$. *Otherwise*, the plan is *sequential*.

Definition 7.2. A planning language L is **Temporally Expressive** iff it can represent problems with required concurrency. *Otherwise*, it is *Temporally Simple*.

7.3 POCL Planning

POCL Planning originates in classical planning [98, 80]. It works on the principle of least commitment, in which commitment on the ordering is delayed until forced by the need to satisfy action preconditions. A state in POCL planning is designed around this notion, defined as $\sigma := \langle Steps, Order, CL, Open\rangle$, in which *Steps* are the actions in the

partial plan, *Order* captures the sequencing of the actions, CL are the causal links of the form $a'[p]a$ stating action a' supports precondition p of a, and *Open* is the set of action preconditions waiting to be supported. The ordering is captured as a constraint $a' \prec a$, specifying that the execution of action a' precedes a. The initial state of the POCL search is given as $\langle\{start, end\}, \{start \prec end\}, \varnothing, \{(g_1, end), \ldots, (g_m, end)\}\rangle$, where start and end are *dummy* actions bringing into effect the *initial* state and enforcing the *goal* conditions $G = \{g_1, \ldots, g_m\}$, respectively.

Typically, POCL search proceeds by branching on a *flaw* and repairing it. Two types of flaws exist: *Open preconditions* and *threats*. *Open preconditions* are fixed by adding a *causal link* of the form $a'[p]a$ to CL, and action a' to *Steps*. *Threats* are actions that can delete the precondition of another action, which are resolved by adding $a' \prec a''$ or $a \prec a'$ for each STRIPS operator $a' \in O$ that threatens a precondition of a that is supported by a''. A state is *invalid* and *terminal* if the ordering is inconsistent. A satisficing solution is found when the state is *valid* and there are no open conditions, $Open = \varnothing$.

7.4 Simple POCL formulation of Temporal Planning

CPT considers a simple extension of temporal planning by adding *time variables* denoting the timestamp of temporal actions, *start* time. A temporal planning problem is $\Pi = \langle F, O, I, G\rangle$, where Π is the STRIPS extension to temporal planning in which $a \in O$ is a durative action consisting of $\mathrm{Pre}_a, \mathrm{Add}_a, \mathrm{Del}_a, \mathrm{Dur}_a$, where Dur_a is the duration of the action, and $\mathrm{Pre}_a, \mathrm{Add}_a, \mathrm{Del}_a$ are suitably defined for the temporal setting. The state σ in POCL planning is redefined to include the time variables as $\sigma' = \langle Steps, Ord', CL, Open, T\rangle$ where $T := \{t_1, \ldots, t_m\}$ is the set of temporal variables where t_i is the *start* time of the action o_i in $Steps := \{o_1, \ldots, o_m\}$. Ord'[1] is slightly modified to capture the temporal constraints over variables T. In the altered state representation, the ordering $o_i \prec o_j$ is captured using a temporal constraint $t_i + \mathrm{Dur}_i < t_j$. The start time associated with the dummy *start* and *end* actions are represented by t_0 and t_1, respectively. The initial state in the temporal extension of POCL planning is $\langle\{start, end\}, \{t_0 \leqslant t_1\}, \varnothing, \{(g_0, end), \ldots, (g_m, end)\}, \{t_0, t_1\}\rangle$.

We have the same types of *flaws* as in the original POCL setting, which are similarly resolved. *Open preconditions* flaws $[p]o_j$ are resolved by adding a causal link $o_i[p]o_j$ to

[1] Henceforth, for simplicity, we use σ and Ord in place of σ' and Ord'.

CL, and adding a temporal constraint $(t_i + \text{Dur}_i < t_j)$ to Ord. If $o_i \notin Steps$, it is added to $Steps$, and a new time variable is created in T. *Threats* to causal link $o_i[p]o_j$ are addressed by either adding a temporal constraint $t_i + \text{Dur}_i < t_j$ or $t_j + \text{Dur}_j < t_i$. The temporal constraints in Ord collectively compose a Simple Temporal Problem (STP) [32], which can be easily verified using the Bellman-Ford algorithm to detect negative cycles in the constraint graph of inequalities.

A search state σ is *valid* if the STP obtained from T is consistent. Furthermore, a solution has been found if the set of open preconditions is empty and no flaws remain. On the other hand, the state σ is *invalid* and *terminal* if the STP is inconsistent. Using the assignments to variable T in the STP's solution with the smallest makespan, a temporal plan can be obtained. The CPT planner encodes this temporal POCL planning formulation into CP, which we present next.

7.4.1 CP Encoding of Temporal POCL Planning

Symbol	Description
t_i	*Start* time of an action i, $i = 1, \ldots, \lvert Steps \rvert$
y_i	Boolean variables capturing *presence of action* o_i in $Steps$ of the terminal state
b_{ij}^p	Boolean variables denoting *active causal links* $o_i[p]o_j$
t_j^p	*Start* time of action supporting *open* condition $[p]o_j$

TABLE 7.1: Quick reference table for the decision variables in the CP model.

We begin by giving a high-level account of the CP encoding of temporal planning from the planner and explain the roles played by the decision variables in Table 7.1. Central to the CPT encoding is the notion of *steps*, $\mathbb{S}$[2] $:= \{o_0, \ldots, o_m\}$, the set of instances of temporal actions in the encoding, $\mathbb{S} \subseteq O \times \mathbb{N}$. $\mathbb{S}$ includes two *dummy* actions for the *start* and *end* actions, o_0 and o_1, respectively. Only a subset of actions in $\mathbb{S}$ will be present in a solution plan. *In-plan* Boolean variables y_i capture the presence of an action in the solution plan. Also, for each action instance, a time variable t_i denotes the *start* time of the action. The set of atoms in the conjunctive precondition formula of an action o_i is represented by Pre_i, and the positive and negative effect atoms of an action by Add_i

[2]We use $[\mathcal{X}]$ to denote the *ordered indexing* of the elements of set $\mathcal{X}$, e.g., $[\mathcal{X}] = (0, \ldots, n)$ for $\mathcal{X} = \{x_0, \ldots, x_n\}$.

and Del_i. We represent the set of *possible* supporters of an atom p with S^p[3], and the *potential threats* that interfere with an atom p by $\bar{S}^p$, both of which can be pre-computed as $S^p := \{a \mid a \in \mathbb{S}, p \in \text{Add}_a\}$ and $\bar{S}^p := \{a \mid a \in \mathbb{S}, p \in \text{Del}_a\}$. Lastly, we create Boolean variables b_{ij}^p enabling us to capture the *active* causal links in the plan, and variable t_j^p to capture the start time of the action supporting the open condition.

Throughout all the encodings discussed in this chapter, we are always concerned with the *makespan* objective. We define it as the time difference between the *start* and the *end* actions, Eq. 7.1. With some trivial modifications, the same encoding could be adopted for minimizing the *total cost* objective.

$$\textbf{Makespan objective: } \min\ t_1 - t_0 \tag{7.1}$$

Constraints. We define the constraints in the CP model of temporal POCL planning, beginning with Constraints 7.2 defined over the *start* and the *end* actions. A fundamental requirement in any solution is that both o_0 and o_1 are in the plan, a condition enforced by Eq. 7.2a. Additionally, executing o_0, which brings the *initial* state into effect, must precede all other actions; similarly, action o_0 must not start before all other actions have finished. Both these conditions are enforced by Eq. 7.2b and Eq. 7.2c. In this context, the CPT planner precomputes the minimal distance between two actions with o_j preceding o_i as $dist(o_j, o_i)$ using h^1_T heuristics[4], an extension of h^1 heuristics for makespan optimal temporal planning. We define δ_{ji} as $\text{Dur}_j + dist(o_j, o_i)$, which allows CPT to specify *tighter* temporal constraints.

$$y_0 \wedge y_1 \tag{7.2a}$$

$$t_0 + \delta_{0i} \leqslant t_i,\ \forall i > 0 \tag{7.2b}$$

$$t_i + \delta i1 \leqslant t_1,\ \forall i > 1 \tag{7.2c}$$

Another essential requirement from the POCL formulation is that the preconditions of every action added to the *Steps* of a state must be inserted into the *Open* precondition, which is encoded by the constraints given in Eq. 7.3. The constraints ensure that all the preconditions of *active* actions, denoted by y_i, are supported by at least one step,

[3]We use $[\mathcal{X}]$ to denote the *ordered indexing* of the elements of set $\mathcal{X}$, e.g. $[\mathcal{X}] = (1, \ldots, n)$ for $\mathcal{X} = \{x_1, \ldots, x_n\}$.

[4]We request the readers to refer to the original CPT paper [152] for a detailed explanation of the h^1_T heuristics and *dist*.

resolving the *open* precondition flaws.

$$y_i \rightarrow \bigwedge_{p \in \mathrm{Pre}_i} \left[\bigvee_{j \in S^q} b^p_{ji} \right] \tag{7.3a}$$

$$b^p_{ji} \rightarrow y_i \wedge y_j \tag{7.3b}$$

When resolving an *open* precondition flaw by adding a causal link, we must also add a temporal constraint, setting the temporal precedence relation between the supporter and the supported action. We describe these constraints in Eq. 7.4.

$$b^p_{ji} \rightarrow [t_j + \delta_{ji} \leqslant t_i] \tag{7.4a}$$

$$b^p_{ji} \rightarrow [t^p_i = t_j] \tag{7.4b}$$

In addition to the *open* condition flaws, we must also resolve the interference situations by *threats* to the causal links. The constraint described in Eq. 7.5 resolves the interference flaw, where $o_i \in S^p$ and $o_j \in S^{\bar{p}}$, specifying a disjunctive constraint over the t_i and t^p_i time variables.

$$y_i \wedge y_j \rightarrow (t_i + \delta_{ij} \leqslant t_j) \vee \left(t_j + \mathrm{Dur}_j + \min_{k \in S^q} dist(o_j, o_k) \leqslant t^q_i \right) \tag{7.5}$$

CPT also specifies *mutex constraints*, described in Eq. 7.6 to address *mutex threats* between actions. It defines *mutex threats* as a pair (o_i, o_j) where the actions interfere, i.e., one deletes the add effects of another, and neither e-deletes each other's preconditions.

$$y_i \wedge y_j \rightarrow (t_i + \delta_{ij} \leqslant t_j) \vee (t_j + \delta_{ji} \leqslant t_i) \tag{7.6}$$

Tightening constraints. In addition to the *necessary* constraints specified above, CPT adds additional tightening constraints to reduce the search space size explicitly. The first set of constraints discussed in Eq 7.7 specify tighter bounds of time variables.

$$t_i \geqslant \min_{j \in [\mathbb{S}], p \in \mathrm{Pre}_i, j \in [S^p]} t_j + \delta_{ij} \tag{7.7a}$$

$$t_i \geqslant t^p_i + \min_{j \in [S^p]} \delta_{ji} \tag{7.7b}$$

$$\min_{j \in [S^p]} t_j \leqslant t^p_i \leqslant \max_{j \in [S^p]} t_j \tag{7.7c}$$

7.5 *Extended* POCL Formulation

The POCL formulation of temporal planning presented in CPT and the resulting CP encoding does not cover all the temporal planning problems specified in PDDL 2.1, a fact noted by the authors themselves [152]. It is also easy to point to this from the CP encoding discussed in Section 7.4.1 as it has no notion of *at start*, *at end*, and *overall* specifications of the durative actions in PDDL 2.1. It appears that the authors consciously considered temporally simple instances in which all the preconditions are *at start* of the action, and all action effects occur at the end, after a time difference equal to the duration of the action. We change this assumption in our *extended POCL formulation of temporal planning*, addressing the cases of required concurrency. For this, we capture additional information about the preconditions and effects in the search state of the POCL planning. The set of *Open* preconditions is denoted by $[p](a, tstamp)$ where p and a take the original meaning of open precondition flaw $[p]a$ and $tstamp \in$ {*at start, at end, overall*} specify the temporal aspects of the action, according to the semantics of PDDL 2.1.

The open precondition flaws $[p](o_j, tstamp)$ in *Open* are resolved by adding a causal link $(o_i, tstamp')[p](o_j, tstamp)$ and adding the action o_i to *Steps* if not already present. Adding the temporal constraints to *Ord* to preserve the sequencing requirement of a causal link is much *harder* for the temporally expressive case as we have to consider the *actual* time stamp of the supported precondition and supporting effect. We do this by considering the {*at start, at end, overall*} preconditions and {*at start, at end*} effects case-by-case. For the case when $tstamp'$ is *at end*, and $tstamp$ is *overall*, we add the constraint $t_i + \text{Dur}_i \leqslant t_j$, as in the CPT formulation. When $tstamp'$ is *at end*, and $tstamp$ is *at start*, we add the constraint $t_i + \text{Dur}_i < t_j$. Similarly, when both, $tstamp'$ and $tstamp$, are *at end*, we add the constraint $t_i + \text{Dur}_i < t_j + \text{Dur}_j$. The temporal constraints when $tstamp'$ is *at start* can be obtained by removing Dur_i from the above constraints.

We must also prevent *threats* from interfering with the causal links that resolve the open conditions. This is also done on a case-to-case basis, the same as above. We provide the details of temporal constraints added into *Ord* to resolve threats in Section 7.6

7.6 Modeling *Extended* POCL Formulation in CP

Symbol	Description
t_i	*Start* time of an action i, $i = 1, \ldots, \lvert Steps \rvert$
y_i	Boolean variables capturing *presence of action* o_i in *Steps* of the terminal state
$b_{ij}^{p[t,t']}$	Boolean variables denoting *active causal links* $(o_i, t)[p](o_j, t')$
$t_j^{p[t]}$	*Start* time of action supporting *open* condition $[p](o_j, t)$

TABLE 7.2: Quick reference table for the decision variables in the CP model.

We use the same set of variables described in Table 7.1, with minor changes, for our *extended* POCL formulation, see Table 7.2. The only difference is an increase in the number of b_{ij}^{p} variables to account for timing information (*at start, at end, and overall*) in the causal links and open preconditions. The primary change in the encoding is the temporal constraints for the updated representation of causal links and associated threats. In addition, we are forced to throw away most of the *tightening* constraints discussed in Section 7.4.1 as they do not generalize to the required concurrency cases. Here too, we optimize for *makespan* as defined in Eq. 7.8.

$$\textbf{Makespan objective:} \ \min \ t_1 - t_0 \tag{7.8}$$

Constraints. A major difference in this encoding is that we do not use the *preprocessing* methods used by the CPT planner to obtain the minimum distance between two actions $dist(a, a')$, as its preprocessing is bound to the assumption of temporally simple problems which we remove to handle required concurrency. Hence, we forgo most of the *tightening* constraints of the original CPT encoding. We only use the minimum distance from the initial state computed using the h_T^1 heuristics, obtaining $dist(o_0, a')$. We set δ_{0i} as $dist(o_0, a')$. We reuse constraints enforcing the basic conditions on the dummy *start* and *end* action in Equations 7.9.

$$y_0 \wedge y_1 \tag{7.9a}$$

$$t_0 + \delta_{0i} \leqslant t_i, \ \forall i > 0 \tag{7.9b}$$

$$t_i + \mathrm{Dur}_i \leqslant t_1, \ \forall i > 1 \tag{7.9c}$$

The requirement of supporters for *open* preconditions is managed by the constraints

described in Equations. 7.10. Here, too, the constraints remain the same as in the original encoding.

$$y_i \rightarrow \bigwedge_{p \in \mathrm{Pre}_i} \left(\bigvee_{j \in S^q} b_{ji}^{p[t,t']} \right) \tag{7.10a}$$

$$b_{ji}^{p[t,t']} \rightarrow y_i \wedge y_j \tag{7.10b}$$

On the other hand, enforcing the temporal constraints associated with the causal links in the POCL formulation of temporally expressive planning is not only different but also quite challenging compared to the original encoding. We present the constraints, case-by-case, in Equations 7.11. Here, we use $\vdash$, $\dashv$, and $\leftrightarrow$, to denote the set of *at start*, *at end*, and *overall* preconditions[5], respectively.

$$b_{ji}^{p[\vdash,\vdash]} \rightarrow (t_j < t_i) \tag{7.11a}$$

$$b_{ji}^{p[\vdash,\leftrightarrow]} \rightarrow (t_j + \leqslant t_i) \tag{7.11b}$$

$$b_{ji}^{p[\vdash,\dashv]} \rightarrow (t_j < t_i + \mathrm{Dur}_i) \tag{7.11c}$$

$$b_{ji}^{p[\dashv,\vdash]} \rightarrow (t_j + \mathrm{Dur}_j < t_i) \tag{7.11d}$$

$$b_{ji}^{p[\dashv,\leftrightarrow]} \rightarrow (t_j + \mathrm{Dur}_j \leqslant t_i) \tag{7.11e}$$

$$b_{ji}^{p[\dashv,\dashv]} \rightarrow (t_j + \mathrm{Dur}_j < t_i + \mathrm{Dur}_i) \tag{7.11f}$$

$$b_{ji}^{p[\vdash,t]} \rightarrow \left(t_i^{p[t]} = t_j \right) \tag{7.11g}$$

$$b_{ji}^{p[\dashv,t]} \rightarrow \left(t_i^{p[t]} = t_j + \mathrm{Dur}_j \right) \tag{7.11h}$$

Similarly, we address the *threats* that can *potentially* interfere with the preconditions on a case-to-case basis, as described in the Equations 7.12, considering each combination of *threats* $(j,t) \in \bar{S}^p$ that delete the atom p and open preconditions $[p](i,t')$, $t \in \{\vdash, \leftrightarrow, \dashv\}$

[5]The semantics of *at start*, *at end*, and *overall* preconditions is as per PDDL 2.1 [49]

and $t' \in \{\vdash, \dashv\}$

$$y_i \wedge y_j \rightarrow (t_i < t_j + \mathrm{Dur}_j) \vee \left(t_j + \mathrm{Dur}_j < t_i^{p[\vdash]}\right),\ (o_j, \dashv) \in \bar{S}^p \tag{7.12a}$$

$$y_i \wedge y_j \rightarrow (t_i + \mathrm{Dur}_i < t_j + \mathrm{Dur}_j) \vee \left(t_j + \mathrm{Dur}_j < t_i^{p[\dashv]}\right),\ (o_j, \dashv) \in \bar{S}^p \tag{7.12b}$$

$$y_i \wedge y_j \rightarrow (t_i + \mathrm{Dur}_i \leqslant t_j + \mathrm{Dur}_j) \vee \left(t_j + \mathrm{Dur}_j \leqslant t_i^{p[\leftrightarrow]}\right),\ (o_j, \dashv) \in \bar{S}^p \tag{7.12c}$$

$$y_i \wedge y_j \rightarrow (t_i < t_j) \vee \left(t_j < t_i^{p[\vdash]}\right),\ (o_j, \vdash) \in \bar{S}^p \tag{7.12d}$$

$$y_i \wedge y_j \rightarrow (t_i + \mathrm{Dur}_i < t_j) \vee \left(t_j < t_i^{p[\dashv]}\right),\ (o_j, \vdash) \in \bar{S}^p \tag{7.12e}$$

$$y_i \wedge y_j \rightarrow (t_i + \mathrm{Dur}_i \leqslant t_j) \vee \left(t_j \leqslant t_i^{p[\leftrightarrow]}\right),\ (o_j, \vdash) \in \bar{S}^p \tag{7.12f}$$

Finally, the Eq. 7.13 encodes the mutex constraints for effect interfering actions.

$$y_i \wedge y_j \rightarrow (t_i < t_j) \vee (t_j < t_i),\ (o_i, \vdash) \in S^p,\ (o_j, \vdash) \in \bar{S}^p \tag{7.13a}$$

$$y_i \wedge y_j \rightarrow (t_i < t_j + \mathrm{Dur}_j) \vee (t_j + \mathrm{Dur}_j < t_i),\ (o_i, \vdash) \in S^p,\ (o_j, \dashv) \in \bar{S}^p \tag{7.13b}$$

$$y_i \wedge y_j \rightarrow (t_i + \mathrm{Dur}_i < t_j) \vee (t_j < t_i + \mathrm{Dur}_i),\ (o_i, \dashv) \in S^p,\ (o_j, \vdash) \in \bar{S}^p \tag{7.13c}$$

$$y_i \wedge y_j \rightarrow (t_i + \mathrm{Dur}_i < t_j + \mathrm{Dur}_j) \vee (t_j + \mathrm{Dur}_j < t_i + \mathrm{Dur}_i),\ (o_i, \dashv) \in S^p,\ (o_j, \dashv) \in \bar{S}^p \tag{7.13d}$$

7.7 Empirical Analysis

Our experiments consist of running a given planner on a temporal planning instance, where the solver process runs on a single CPU core (Intel Xeon running at 2GHz). We impose resource usage limits on each run, 1800 seconds on runtime, and 8 GBytes on memory. We solve our CP model using Google's CpSat solver, which provides typical CP constraints out of the box. Minor modifications are made to the CP encoding to accommodate the API of Google's CpSat solver. Also, we note that the temporal action set in the encoding, $\mathbb{S}$, was bootstrapped with a single instance of each operator in the *grounded* representation of PDDL 2.1.

We compare the performance of CpSat solving our model with notable optimal and satisficing temporal planners. The former includes, in no particular order, Temporal Fast Downward (TFD) [42], an extension of Fast Downward, which was the runner-up in IPC 2011 and 2014, the TPSHE [79] planner, which addresses the *required concurrency* concern using classical planning but is only able to cover a subset of temporally expressive

cases, and two forward-chaining planners POPF [25] and OPTIC [8], both of which are claimed to be *complete* for temporal planning, covering all the cases of *required concurrency*. All *optimal* planners were configured to *minimize* the *makespan*.

For *satisficing* planning, we additionally compare ITSAT [118], which is conceptually closer to CpSat than other baseline planners. ITSAT follows the strategy known as Logic-based Benders Decomposition [75], decomposing the temporal planning problem into a SAT master and a sub-problem that takes the form of a Simple Temporal Problem (STP) [32]. When the STP is inconsistent, i.e., it has negative cycles, a procedure adds an explanatory clause to the SAT master, removing a set of previously valid parallel atemporal plans. We also include TAMER [147] in the set of baseline *satisficing* planners. TAMER is a recent Temporal Planner, included in the Unified Planning Framework (UPF) [102], part of a larger AIPlan4EU[6] initiative to deliver on-demand AI Planning services to Europe, which makes it attractive to study.

We evaluate the planners on IPC 2011, 2014, and 2018 instances. This enables us to test the planners on various cases with different structures, including both *simple* and *required concurrency* domains. Additionally, in *match-cellar-14*, we modified the PDDL action schemas to break symmetry by *sequencing* the usages of *matches* and *fuses*, making it different from *match-cellar-11*.

Optimizing Planning. We first discuss the performance of the planners in terms of *optimal coverage*, which is the count of instances for which the planner successfully proves *optimality*. From Table 7.3, we observe that while, overall, none of the planners scale up when it comes to searching for optimal plans, our *extended* CpSat encoding performs remarkably well on domains with *required concurrency*. The results are particularly notable for the *Cushing* domain that has complex dependencies between three action schemas, requiring them to overlap in a particular manner. We note that the only other *complete* planner OPTIC runs out of memory during the search for the vast majority of Cushing instances before finding an optimal solution. Our planner also performs well on the *temporal machine shop* (TMS) instances. In comparison, no planner performs well on the two *required concurrency* domains, *match-cellar* and *turn-and-open*, which can be attributed to the existence of symmetrical solutions and long *sequential* dependencies. The *match-cellar* domain requires that a fuse is fixed while the match is lit, hence, it requires the *fix-fuse* action to run while the *light-match* action is executing. However, the

[6] https://www.ai4europe.eu/ai-community/projects/aiplan4eu

IPC-2014, 2018 Optimizing Solution						
Domain	*Count*	CpSat	TFD	TPSHE	OPTIC	POPF
Required Concurrency Domains						
matchcellar-11	20	1	1	4	0	0
matchcellar-14	20	0	0	0	0	0
tms-11	20	20	0	0	0	0
tms-14	20	20	0	0	0	0
cushing-18	10	10	0	0	2	0
turn-and-open-11	20	0	0	0	0	0
turn-and-open-14	20	0	0	0	0	0
Total (*Req. Conc.*)	*130*	**51**	1	4	8	2
Temporally Simple Domains						
crew-planning-11	20	5	0	4	0	0
elevator-11	20	0	0	0	0	0
floor-tile-11	20	0	0	5	0	0
openstacks-11	20	0	0	0	0	0
parc-printer-11	20	1	0	0	0	0
parking-11	20	0	0	0	0	0
peg-solitaire-11	20	18	10	18	6	7
sokoban-11	20	0	2	4	0	0
storage-11	20	0	0	0	0	0
driver-log-14	20	0	0	0	0	0
floor-tile-14	20	0	0	0	0	0
map-analyzer-14	20	0	0	0	0	0
parking-14	20	0	0	0	0	0
road-traffic-14	20	0	0	0	0	0
storage-14	20	0	0	0	0	0
satellite-14	20	0	0	0	0	0
airport-18	10	7	5	8	2	0
floortile-18	10	0	0	0	0	0
mapanalyser-18	10	0	0	0	0	0
parking-18	10	0	0	0	0	0
quantum-circuit-18	10	0	2	5	2	0
road-traffic-18	10	0	0	0	0	0
sokoban-18	10	0	1	2	0	0
trucks-18	10	1	1	3	0	0
Total (*T. Simple*)	*400*	32	21	**49**	10	7
Total	*530*	**83**	22	53	12	7

Table 7.3: *Optimizing* Planing Results.

domain assumes that exactly one agent is repairing the fuses, hence requiring sequencing of all *fix-fuse* actions. *Turn-and-open*, similarly, is a domain with mixed *required concurrency* and *sequencing* requirements.

Our planner has shown mixed results in *optimizing* temporally simple instances, most of which are adopted from classical planning, requiring planners to find long sequencing of actions. This is the Achilles heel of our CP encoding. Though quite performant on optimal scheduling benchmarks, the default *activity-based* heuristics used by CpSat do not seem to scale on instances requiring long action sequences. This is similar to what we observe in our work on CP encodings of Lifted Sequential Planning in Chapter 5, pointing to a common challenge in solving CP encodings of planning with general purpose solvers.

The TPSHE planner performs best in these instances. However, it cannot handle all cases of required concurrency, including *Cushing* instances. The TFD planner, another planner that repurposes the classical planning approaches for temporal planning, shows the next best performance, strengthening our belief that sequencing actions is an important and challenging aspect of most temporal planning benchmark instances.

	IPC-2014, 2018 Satisficing Solution							
Domain	*Count*	CpSat	TFD	TPSHE	OPTIC	POPF	ITSAT	TAMER
Required Concurrency Domains								
match-cellar-11	20	1	20	20	6	20	20	2
match-cellar-14	20	0	1	0	16	15	0*	0*
tms-11	20	20	0	12	5	6	20	0*
tms-14	20	20	0	7	0	1	20	0*
turn-and-open-11	20	0	19	20	9	9	6	0*
turn-and-open-14	20	0	18	19	10	9	6	0*
Cushing-18	10	10	0	0	10	10	0	1
Total (*Req. Conc.*)	*130*	51	58	**78**	56	70	72	3
Temporally Simple Domains								
quantum-circuit-18	10	0	8	10	8	8	5	5
crew-planning-11	20	9	20	20	20	20	20	0*
elevator-11	20	0	20	20	2	2	0	0*
floor-tile-11	20	5	5	7	0	0	20	0*
openstacks	20	0	20	20	20	20	0	0*
parc-printer-11	20	1	10	6	0	0	18	0
parking-11	20	0	19	20	20	20	0	0
peg-solitaire-11	20	18	19	20	19	19	20	15
sokoban-11	20	0	3	12	4	4	6	0*
storage-11	20	0	0	7	0	0	0*	0*
driver-log-14	20	0	0	18	0	0	2	0*
floor-tile-14	20	0	0	3	0	0	20	0*
map-analyzer-14	20	0	16	20	0	0	0	0*
parking-14	20	0	20	20	14	13	12	0
road-traffic-14	20	0	0	16	0	0	0	0*
storage-14	20	0	0	9	0	0	0*	0*
satellite-14	20	0	17	18	4	3	0*	0
airport-18	10	8	9	9	3	3	8	3
floortile-18	10	0	0	1	0	0	10	0
mapanalyser-18	10	0	8	10	0	0	0	0*
parking-18	10	0	10	10	7	7	6	0
road-traffic-18	10	0	0	7	0	0	0	0*
sokoban-18	10	0	1	6	1	1	3	0*
trucks-18	10	1	10	10	8	10	0*	0
Total (*T. Simple*)	*400*	42	215	**299**	130	130	150	23
Total	*530*	93	273	**377**	186	200	222	26

Table 7.4: *Satisficing* Planning Results. *'*' indicates domains where the planner failed to instantiate the planner due to runtime error when loading the PDDL representation.*

Satisficing Planning. While our CpSat encoding is the best performer in *optimizing*, it lacks significantly in *satisficing* planning, especially on the instances for which valid plans include long sequences of actions. This points to a lack of proper search guidance, which may allow the planner to find an upper bound quickly. All baseline planners, except TAMER, perform better than our encoding in the *satisficing* planning. TPSHE

leads the rankings with 100 additional solved tasks compared to the nearest competitor TFD, which solves 273 instances out of 530. This points to the advantage of strategies that sequence the actions well, driven by classical planning strategies. ITSAT also performed well and solved more problems than OPTIC and POPF. It failed to process the PDDL on a few domains and, more importantly, returned invalid plans for 5 instances of *Mapanalyser* and 3 instances of *Road-traffic*. This suggests there are underlying problems with the implementation or perhaps with the problem decomposition, indicating that an in-depth analysis of the ITSAT planner is required, which is beyond the scope of this thesis. Finally, TAMER failed to parse and translate many PDDLs, which are indicated with a '*' in Table 7.4, significantly reducing its coverage. Furthermore, it returned *invalid plans* for 6 *Parking* instances and 1 *Trucks* instances. These issues suggest that TAMER is not fully compatible with PDDL.

7.8 Conclusion

We show that the CP approach to temporal planning can handle instances with *required concurrency* described in temporally expressive languages. Furthermore, the CP encoding of *extended* temporal POCL planning is quite performant on benchmark instances with the primary concern of *concurrency*, particularly the *Cushing* instances that are difficult or infeasible for other planners. However, the advantage quickly disappears with the degree of sequencing required in plans. The CpSat solver struggles to find a solution to these instances. This leaves both an open question and a *potential* research question on customizing the CP search strategies and encodings to address the concern of sequencing better, allowing the CP solver to find an upper bound quickly. Another concern that requires attention is symmetry breaking, which is important in *optimal* temporal planning, as many planning instances like *match-cellar* and *turn-and-open* have large tracts of search space that are symmetrical, making it difficult to improve the lower bound. We note that automated symmetry detection methods used by CpSat did not help in these instances.

Part IV

Conclusion

"To succeed in life and achieve results, you must understand and master three mighty forces— desire, belief, and expectation." Iyadurai."

A.P.J. Abdul Kalam

8

Conclusions and Future Work

In this chapter, we present a synopsis of our contributions, their limitations, and future work in the direction we pursued here. We highlight the challenges we encountered, the things that worked and did not work, and the promising paths for future research that we did not have time to follow but wish to see happen.

8.1 Summary

Our research explored many approaches to solve two classes of deterministic planning problems, sequential and temporal, addressing the challenges posed by large-scale planning problems, dealing with concerns of high width in width-based search algorithms and scalability concerns in large problems with a significant number of objects and large complex systems of constraints. We also tackled the concern of required concurrency in temporal planning. We explored the paradigm of *planning as search* and *approximation approaches* from stream computing in Internet-Of-Things (IOT), the knowledge of which enabled us to design tractable approximation of novelty search, allowing us to

handle computations in problems with high state novelty values, which were previously impractical.

We also surveyed the field of *Constraint Satisfaction Problems* (CSPs), focusing not only on the Constraint Programming (CP) approach to CSPs but also on sub-fields of Mixed Integer Linear Programming (MILP) and Boolean Satisfiability (SAT), looking for methods and techniques in those areas that intersect in some way with the planning problems. We also explored the possibility of using decomposition approaches within the Logic-based and Generalized Benders Decomposition frameworks. We eventually pursued the Lazy Clause and Constraint generation (LCG) framework to address the scalability issues arising from the need to define explicit constraints in memory, where planning models usually have several thousands of constraints. LCG, which implicitly allows us to represent large tracts of complex constraints using propagators, presents a way to address this.

Overall, within all the branches we explored in our research, a few resulted in remarkable success, many gave mixed results, and a few failed despite all the effort. We note that the state-of-the-art CSP-solving technology as of 2023 could not compare against the established methods in *planning as heuristics search*, in *satisficing* setting, both in sequential and temporal planning instances of the IPCs. In our assessment, the CP planners we developed struggled with instances requiring a long chain of actions with sequential dependencies in the plan. This is still significant, as we acquired valuable insights from this work, including opportunities for improvement, which we believe is a successful outcome. We discuss our insights in more detail in the following sections.

Chapter Outline. We give an overview of our work in parts, starting with the impact and challenges of our work on Tractable Approximation of Novelty Search, followed by the merit of our work in Sequential Planning with LCG and its limitations, and concluding with an account of the importance and restriction of our work on CP Approach to Temporal Planning with *required concurrency*.

8.2 Approximate Novelty Search

The computation of *state novelty* is integral to the width-based search algorithms. However, its runtime and space computation is exponential in the novelty values, becoming

impractical to compute for value > 2 for all except tiny planning instances. The primary contribution of our approach is the *approximation* of state novelty, which reduces the complexity from *exponential* to *linear* in the size of the instance. We address the challenge by designing *approximations* based on *random sampling* and *Bloom filters*, trading off measurement accuracy for computational guarantees.

The novelty approximation has the potential to impact all works using width-based search. The approach only requires the factored representation of a state as input and can, hence, be easily incorporated into other variants.

Limitations. We *trade-off accuracy for computational guarantees* in our Sampling and Bloom Filter based *approximation* of novelty measure. This trade-off follows a theoretical probability of error, which we prove in Section 4.3. That is to say, the *accuracy* of *Bloom filters* in representing the relational predicate capturing the existence of a tuple of atoms *decreases* as the ratio of tuples to the Bloom filter size increases. Similarly, the accuracy of inference made using a random sample of tuples instead of an exhaustive set depends on the sample size. Hence, a main limitation of the approach is the low accuracy of novelty approximation for states with high novelty values, where the number of tuple combinations that we need to evaluate for exact computation is significant, much larger than the sample and Bloom filter size. While these methods are parameterized by the sample size and size of the Bloom filters, which can be modified to fine-tune the accuracy, typically, the amount of memory available is fixed, and a sample of tuples that is cubic in the size of the problem is already too big and impractical to enumerate, except for tiny instances.

Future Work. With limitations come opportunities. One such option is to sample from a weighted collection of tuples instead of choosing randomly. This would lower the frequency at which *distinct* tuples are observed by the Bloom filter, as tuples with higher weights are more probable to be selected, increasing its accuracy. The key challenge here is prioritizing and assigning weights to the tuples, which remains an open question. However, possibilities exist as only a subset of tuples are relevant to admitting a particular sequence of actions in the novelty search. This leaves an option of discarding the irrelevant tuples, assuming they have been seen in the initial state. Even if this information is available with low confidence, the latter could be assigned lower weights than the former and used in approximate novelty with weighted sampling.

Additionally, we have only looked at specific approximate methods from the field of *stream computing*, and we note that other approaches to compute the novelty measure efficiently and accurately may exist. So, that line of thinking remains open as well.

8.3 Lifted Sequential Planning with Lazy Clause and Constraint Generation

This work demonstrates the opportunities the lazy clause generation approach to constraint programming opens up, including user-propagators for planning. We present a CP encoding of Lifted Sequential Planning that by itself performs very well on *hard-to-ground* benchmarks when implemented in CpSat. However, it improves further when paired with a planning-specific propagator that implements the *persistence* constraints, improving the LCG CDCL's scalability. The resulting planner scales very well in problems with large numbers of objects and numeric domains, including the multi-modal project scheduling problems (MMPSP) with numeric time domain.

Additionally, we demonstrate that the planning as search strategies, including depth-first search and novelty search, can be integrated into the framework of LCG CDCL. For this, we extended the CpSat solver with programmed search heuristics emulating depth-first search, enabling us to implement the notion of novelty given depth as a user-propagator efficiently. While the work only slightly improves the performance of the Lifted CP encoding, it highlights that the LCG CDCL framework is general enough to capture interesting search paradigms in planning.

Limitations. Our causal encoding performs well on problems with small plan lengths and scales well in situations with large numbers of objects and numeric domains. However, our approach is not suited for problems with long plans. There are multiple *potential* reasons for it, including the number of variables in the encoding, which follows a quadratic growth rate along with plan length, the implementation of *element* constraints by CpSat, and the choice of search space explored by *activity-based* heuristics. CpSat reduces the *element* constraints to equalities, transforming them into inequalities encoded in a *precedences propagator*, which implements integer difference logic using the Bellman-Ford algorithm. While this helps us in problems with numeric domains, like *MMPSP*, there is no clear advantage of the approach when the instance size is smaller

than what is seen in the *hard-to-ground* instances, and the size of the universe does not impact the scalability of the planner.

Another limitation of our planner is that it only optimizes plan length and does not consider plan costs. General (positive) action costs can be added to the current encoding by implicitly representing functions mapping ground actions to costs from the assignments to the act_i and arg_{ij} variables as integer variables bound by element constraints, then setting the objective to minimize their accumulated sum. However, we note that the search for "globally" optimal plans cannot end with the first cost-optimal plan found when solving CSPs: $T(P, k_1), T(P, k_2)...T(P, k_i), k_i - k_{i-1} >= 1$, sequentially, as there may exist a plan with smaller cost but inducing a trajectory in the transition system with more steps. Algorithms for searching optimal plans in this setting are an open research question.

Future Work. Our work on integrating planning approaches into the LCG CDCL framework, including planning-specific propagators for *persistence constraints* and *novelty bound*, highlights a novel avenue of research in planning. We believe that the encoding performance is tightly coupled to procedures implementing constraints, and significant opportunities exist to improve performance by designing and incorporating planning-specific propagators and heuristics.

8.4 CP Approach to Temporal Planning with *Required Concurrency*

Our work extends Temporal POCL Planning to address concurrency concerns in instances with *required concurrency*, taking direct inspiration from CPT's [152] POCL encoding, which made some simplifying assumptions and did not precisely address the STRIPS fragment of PDDL 2.1. Our planner can find *makespan* optimal plans for instances with *required concurrency* and performs very well on such problems. This is significant as not many planners can handle all classes of temporal planning problems with different structures of dependencies between actions in the plan.

Limitations. Our CP encoding of extended Temporal POCL Planning has two noteworthy limitations. One, it does not scale well on problems that require planners to sequence many actions to find a plan. For example, consider the problems of *temporal*

machine shop and *match cellar*, both of which have *required concurrency* that our planner handles well. However, while the planner solves all the instances of *temporal machine shop*, it can only solve one of *match cellar*. The planner struggles in both optimizing and satisficing planning in this domain. Here, the difference between the two domains lies in the sequential dependencies between the plan's actions. In *match cellar*, it is assumed that there is only one agent. Hence, it requires that *fuses* are fixed one at a time, sequentially. In comparison, the *temporal machine shop* instances have minimum sequencing requirements. The same concern exists in the *turn-and-open* instances where our planner is unable to solve any instances.

Another limitation is that the encoding requires the planner to estimate a set of operator instances from the collection of durative operators in the problem description and then construct the CP model that only allows those instances in the plan. This set is unknown beforehand. Choosing too many action instances would increase the encoding size, adding much overhead to the solving process. We configured the planner to select one instance of each action. However, this limits the planner to instances with at most one copy of each operator in the plan. Choosing the number of instances in this setting remains an open question.

Future Work. While the planner performs well in optimizing planning, it struggles to find an upper bound on makespan (satisficing plan) quickly. This highlights the need to adopt the CDCL heuristics to a planning-specific one, much like the planning as satisfiability heuristic [125] for classical planning. Also, more work is required to improve lower bounds quickly in problems like *match cellar* where large tracts of symmetrical plans exist. While many baseline planners successfully find an upper bound, they struggle to prove optimality due to the symmetry.

Bibliography

[1] Mohammad Abdulaziz, Charles Gretton, and Michael Norrish. A State-Space Acyclicity Property for Exponentially Tighter Plan Length Bounds. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 27, pages 2–10, 2017.

[2] Vidal Alcázar and Alvaro Torralba. A Reminder about the Importance of Computing and Exploiting Invariants in Planning. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 25, pages 2–6, 2015.

[3] Fahiem Bacchus and Froduald Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial intelligence*, 116(1-2):123–191, 2000.

[4] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ Planning. *Computer Intelligence*, 11(4):625–655, 1995.

[5] José L. Balcázar. The Complexity of Searching Implicit Graphs. *Artificial Intelligence Journal*, 86(1):171–188, 1996.

[6] Wilmer Bandres, Blai Bonet, and Héctor Geffner. Planning with Pixels in (Almost) Real Time. In *Proc. of the AAAI Conference (AAAI)*, volume 32, 2018.

[7] Hachemi Bennaceur. A Comparison between SAT and CSP Techniques. *Constraints*, 9:123–138, 2004.

[8] J Benton, Amanda Coles, and Andrew Coles. Temporal Planning with Preferences and Time-Dependent Continuous Costs. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 22, pages 2–10, 2012.

[9] Dmitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 4th edition, 2017.

[10] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CaDiCal, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. *SAT Competition*, 2020:50, 2020.

[11] Burton H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[12] Avrim L. Blum and Merrick L. Furst. Fast Planning through Planning Graph Analysis. In *Proc. of Int'l Joint Conference on Artificial Intelligence (IJCAI)*, pages 1636–1642, 1995.

[13] Blai Bonet and Héctor Geffner. HSP: Heuristic Search Planner. *AIPS-98 Planning Competition*, pages 60–72, 1998.

[14] Blai Bonet and Héctor Geffner. Planning as Heuristic Search: New Results. In *Proc. of European Conference in Artificial Intelligence (ECAI)*, pages 360–372, 1999.

[15] Blai Bonet and Héctor Geffner. Planning as Heuristic Search. *Artificial Intelligence Journal*, 129(1–2):5–33, 2001.

[16] Nicholas Bourbaki. *Elements of Mathematics: Theory of Sets*. Springer-Verlag, 1968.

[17] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer Set Programming at a Glance. *Communications of the ACM*, 54(12):92–103, 2011.

[18] Andrei Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485–509, 2004.

[19] Ethan Burns, Matthew Hatem, Michael J. Leighton, and Wheeler Ruml. Implementing Fast Heuristic Search Code. In *Proc. of the Annual Symposium on Combinatorial Search*, volume 3, pages 25–32, 2012.

[20] Tom Bylander. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence Journal*, 69(1-2):165–204, 1994.

[21] Isabel Cenamor, Tomás de la Rosa, and Fernando Fernández. IBACOP and IBACOP2 planner. *IPC-8 planner abstracts*, pages 35–38, 2014.

[22] Yair Censor. Pareto Optimality in Multiobjective Problems. *Applied Mathematics and Optimization*, 4(1):41–59, 1977.

[23] Partha P. Chakrabarti, Sujoy Ghose, Arup Acharya, and S.C. De Sarkar. Heuristic Search in Restricted Memory. *Artificial Intelligence Journal*, 41(2):197–221, 1989.

[24] Yixin Chen, Ruoyun Huang, Zhao Xing, and Weixiong Zhang. Long-distance Mutual Exclusion for Planning. *Artificial Intelligence Journal*, 173(2):365–391, 2009.

[25] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-Chaining Partial-Order Planning. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 20, pages 42–49, 2010.

[26] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. COLIN: Planning with Continuous Linear Numeric Change. *Journal of Artificial Intelligence Research*, 44:1–96, 2012.

[27] Andrew Coles, Maria Fox, Derek Long, and Amanda Smith. Planning with Problems Requiring Temporal Coordination. In *Proc. of the AAAI Conference (AAAI)*, pages 892–897, 2008.

[28] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, pages 143–152. Association for Computing Machinery, 2023.

[29] Augusto B. Corrêa, Florian Pommerening, Malte Helmert, and Guillem Francès. Lifted Successor Generation Using Query Optimization Techniques. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 30, pages 80–89, 2020.

[30] Augusto B. Corrêa and Jendrik Seipp. Best-First Width Search for Lifted Classical Planning. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 32, pages 11–15, 2022.

[31] William Cushing, Subbarao Kambhampati, and Daniel S. Weld. When is Temporal Planning Really Temporal? In *Proc. of Int'l Joint Conference on Artificial Intelligence (IJCAI)*, pages 1852–1859, 2007.

[32] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal Constraint Networks. *Artificial Intelligence Journal*, 49(1-3):61–95, 1991.

[33] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[34] Austin J. Dionne, Jordan T. Thayer, and Wheeler Ruml. Deadline-Aware Search Using On-Line Measures of Behavior. In *Proc. of the Annual Symposium on Combinatorial Search*, volume 2, pages 39–46, 2011.

[35] Minh Binh Do and Subbarao Kambhampati. Planning as Constraint Satisfaction: Solving the Planning Graph by compiling it into CSP. *Artificial Intelligence Journal*, 132(2):151–182, 2001.

[36] Minh Binh Do and Subbarao Kambhampati. Sapa: A Multi-objective Metric Temporal Planner. *Journal of Artificial Intelligence Research*, 20:155–194, 2003.

[37] Stefan Edelkamp. Taming Numbers and Durations in the Model Checking Integrated Planning System. *Journal of Artificial Intelligence Research*, 20:195–238, 2003.

[38] Stefan Edelkamp and Stefan Schrödl. *Heuristic Search – Theory and Applications*. Morgan Kauffmann, 2012.

[39] Salomé Eriksson and Malte Helmert. Certified Unsolvability for SAT Planning with Property Directed Reachability. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 30, pages 90–100, 2020.

[40] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic SAT-Compilation of Planning Problems. In *Proc. of Int'l Joint Conference on Artificial Intelligence (IJCAI)*, volume 97, pages 1169–1176, 1997.

[41] Kutluhan Erol, Dana S. Nau, and Venkatramana S. Subrahmanian. Complexity, Decidability and Undecidability results for Domain-independent Planning. *Artificial Intelligence Journal*, 76(1-2):75–88, 1995.

[42] Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. Using the Context-Enhanced Additive Heuristic for Temporal and Numeric Planning. In *Towards Service Robots for Everyday Environments: Recent Advances in Designing Service Robots for Complex Tasks in Everyday Environments*, pages 49–64. Springer, 2012.

[43] Sam Ezersky. Letter Boxed — New York Times Logic Puzzles. https://www.nytimes.com/puzzles/letter-boxed. Accessed: 2023-03-19.

[44] Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. IPASIR-UP: User Propagators for CDCL. In *Proc. of the Int'l Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2023.

[45] Richard E. Fikes and Nils J. Nilsson. STRIPS, A Retrospective. *Artificial Intelligence Journal*, 59(1-2):227–232, 1993.

[46] Daniel Fišer. CPDDL: A library for Automated Planning. https://gitlab.com/danfis/cpddl. Accessed: 2022-04-20.

[47] Daniel Fišer and Florian Pommerening. International Planning Competition 2023 – Classical Tracks. https://ipc2023-classical.github.io/. Accessed: 2023-09-19.

[48] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[49] Maria Fox and Derek Long. Modelling Mixed Discrete-Continuous Domains for Planning. *Journal of Artificial Intelligence Research*, 27:235–297, 2006.

[50] Guillem Francès. *Effective Planning with Expressive Languages*. PhD thesis, DTIC, 2017.

[51] Guillem Francès and Héctor Geffner. Modeling and Computation in Planning: Better Heuristics from More Expressive Languages. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 25, pages 70–78, 2015.

[52] Guillem Francès, Miquel Ramirez, and Collaborators. Tarski: An AI planning modeling framework. https://github.com/aig-upf/tarski. Accessed: 2020-09-11.

[53] Kathryn Francis, Jorge Navas, and Peter J. Stuckey. Modelling Destructive Assignments. In *Int'l Conference on Principles and Practice of Constraint Programming (CP)*, pages 315–330, 2013.

[54] Santiago Franco, Alvaro Torralba, Levi H.S. Lelis, and Mike Barley. On Creating Complementary Pattern Databases. In *Proc. of Int'l Joint Conference on Artificial Intelligence (IJCAI)*, pages 4302–4309, 2017.

[55] Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo: A New Grounder for Answer Set Programming. In *Proc. of the Int'l Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 266–271, 2007.

[56] Héctor Geffner. Planning Graphs and Knowledge Compilation. In *Proc. of Principles of Knowledge Representation and Reasoning (KR)*, volume 4, pages 662–672, 2004.

[57] Héctor Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning.* Morgan & Claypool Publishers, 2013.

[58] Héctor Geffner and Patrik Haslum. Admissible Heuristics for Optimal Planning. In *Proc. of AIPS*, pages 140–149, 2000.

[59] Alfonso Gerevini and Ivan Serina. LPG: A Planner Based on Local Search for Planning Graphs with Action Costs. In *Proc. of AIPS*, volume 2, pages 281–290, 2002.

[60] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Danie Weld, and David Wilkins. PDDL: The Planning Domain Definition Language. Technical report, Yale University, 1998.

[61] Nicola Gigante, Andrea Micheli, Angelo Montanari, and Enrico Scala. Decidability and Complexity of Action-Based Temporal Planning over Dense Time. In *Proc. of the AAAI Conference (AAAI)*, volume 34, pages 9859–9866, 2020.

[62] Cordell Green. Application of Theorem Proving to Problem Solving. In *Proc. of Int'l Joint Conference on Artificial Intelligence (IJCAI)*, 1969.

[63] Andrew R. Haas. The Case for Domain-Specific Frame Axioms. In *The Frame Problem in Artificial Intelligence*, pages 343–348, 1987.

[64] Keith Halsey, Derek Long, and Maria Fox. CRIKEY - A Temporal Planner Looking at the Integration of Scheduling and Planning. In *Workshop on Integrating Planning into Scheduling, ICAPS*, pages 46–52, 2004.

[65] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[66] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*. Morgan & Claypool Publishers, 2019.

[67] Malte Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[68] Malte Helmert. Concise Finite-Domain Representations for PDDL planning tasks. *Artificial Intelligence Journal*, 173(5–6):503–535, 2009.

[69] Malte Helmert and Carmel Domshlak. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 19, pages 162–169, 2009.

[70] Malte Helmert and Héctor Geffner. Unifying the Causal Graph and Additive Heuristics. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, pages 140–147, 2008.

[71] Malte Helmert, Gabriele Röger, Jendrik Seipp, Erez Karpas, Jörg Hoffmann, Emil Keyder, Raz Nissim, Silvia Richter, and Matthias Westphal. Fast Downward Stone Soup. *IPC-7 planner abstracts*, pages 38–45, 2011.

[72] Joerg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered Landmarks in Planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[73] Jörg Hoffmann and Bernhard Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[74] John N. Hooker. *Integrated Methods for Optimization*. Springer, 2012.

[75] John N. Hooker and Greger Ottosson. Logic-based Benders Decomposition. *Mathematical Programming*, 96(1):33–60, 2003.

[76] Daniel Höller and Gregor Behnke. Encoding Lifted Classical Planning in Propositional Logic. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 32, pages 134–144, 2022.

[77] International planning competitions – classical tracks. https://www.icaps-conference.org/competitions/. Accessed: 2019-10-19.

[78] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proc. of the ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119, 1987.

[79] Sergio Jiménez, Anders Jonsson, and Héctor Palacios. Temporal Planning with Required Concurrency Using Classical Planning. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 25, pages 129–137, 2015.

[80] Subbarao Kambhampati, Craig A. Knoblock, and Qiang Yang. Planning as Refinement Search: A Unified Framework for Evaluating Design Tradeoffs in Partial-Order Planning. *Artificial Intelligence Journal*, 76(1-2):167–238, 1995.

[81] Michael Katz, Nir Lipovetzky, Dany Moshkovich, and Alexander Tuisov. Adapting Novelty to Classical Planning as Heuristic Search. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, pages 172–180, 2017.

[82] Michael Katz, Shirin Sohrabi, Horst Samulowitz, and Silvan Sievers. Delfi: Online Planner Selection for Cost-Optimal Planning. *IPC-9 planner abstracts*, pages 57–64, 2018.

[83] Henry Kautz, David McAllester, and Bart Selman. Encoding Plans in Propositional Logic. In *Proc. of Principles of Knowledge Representation and Reasoning (KR)*, volume 96, pages 374–384, 1996.

[84] Henry Kautz and Bart Selman. Planning as Satisfiability. In *Proc. of European Conference in Artificial Intelligence (ECAI)*, volume 92, pages 359–363, 1992.

[85] Rainer Kolisch and Arno Sprecher. PSPLIB—a project scheduling problem library. *European journal of operational research*, 96(1):205–216, 1996.

[86] Richard E. Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence Journal*, 27(1):97–109, 1985.

[87] Andrei Krokhin and Stanislav Živný. *The Constraint Satisfaction Problem: Complexity and Approximability*. Schloss Dagstuhl, 2017.

[88] François Laburthe and OCRE Project Team. CHOCO: Implementing a CP Kernel. In *Proc. of TRICS: Techniques for Implementing Constraint Programming Systems*, volume 55, pages 71–85, 2000.

[89] Pascal Lauer, Alvaro Torralba, Daniel Fišer, Daniel Höller, Julia Wichlacz, and Jörg Hoffmann. Polynomial-Time in PDDL Input Size: Making the Delete Relaxation Feasible for Lifted Planning. In *Proc. of Int'l Joint Conference on Artificial Intelligence (IJCAI)*, pages 4119–4126, 2021.

[90] Vladimir Lifschitz. On the semantics of STRIPS. In *Reasoning about Actions and Plans: Proc. of the 1986 Workshop*, pages 1–9, 1987.

[91] Nir Lipovetzky and Héctor Geffner. Width and Serialization of Classical Planning Problems. In *Proc. of European Conference in Artificial Intelligence (ECAI)*, pages 540–545, 2012.

[92] Nir Lipovetzky and Héctor Geffner. A Polynomial Planning Algorithm that Beats LAMA and FF. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 27, pages 195–199, 2017.

[93] Nir Lipovetzky and Héctor Geffner. Best-First Width Search: Exploration and Exploitation in Classical Planning. In *Proc. of the AAAI Conference (AAAI)*, volume 31, 2017.

[94] Nir Lipovetzky, Miquel Ramirez, Christian Muise, and Héctor Geffner. Width and Inference Based Planners: SIW, BFS(f), and PROBE. In *Int'l Planning Competition (IPC)*, page 43, 2014.

[95] Panos Louridas. *Real-World Algorithms: A Beginner's Guide*. MIT Press, 2017.

[96] F. Maris and P. Régnier. TLP-GP: New Results on Temporally-Expressive Planning Benchmarks. In *IEEE International Conference on Tools with Artificial Intelligence*, volume 1, pages 507–514, 2008.

[97] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT press, 1998.

[98] David McAllester and David Rosenblitt. Systematic Nonlinear Planning. In *Proc. of the AAAI Conference (AAAI)*, 1991.

[99] John McCarthy. Formalization of STRIPS in situation calculus. Technical report, 1985.

[100] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1969.

[101] Drew McDermott. The Formal Semantics of Processes in PDDL. In *Proc. of ICAPS Workshop on PDDL*, pages 101–155, 2003.

[102] Andrea Micheli, Alexandre Arnold, and Arthur Bit-Monnot. Unified planning: A python library making planning technology accessible. In *International Conference on Automated Planning and Scheduling, System Demonstration*, 2022.

[103] Hootan Nakhost, Jörg Hoffmann, and Martin Müller. Resource-Constrained Planning: A Monte Carlo Random Walk Approach. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 22, pages 181–189, 2012.

[104] Alexander Nareyek, Eugene C. Freuder, Robert Fourer, Enrico Giunchiglia, Robert P. Goldman, Henry Kautz, Jussi Rintanen, and Austin Tate. Constraints and AI Planning. *IEEE Intelligent Systems*, 20(2):62–72, 2005.

[105] Robert Nieuwenhuis and Albert Oliveras. DPLL(T) with Exhaustive Theory Propagation and Its Applications to Difference Logic. In *Proc. of the Int'l Conf. on Computer Aided Verification (CAV)*, pages 321–334, 2005.

[106] Nils Nilsson and Richard Fikes. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence Journal*, 1:27–120, 1971.

[107] Sergio Núnez, Daniel Borrajo, and C. Linares López. MIPlan and DPMPlan. *IPC-8 planner abstracts*, pages 13–16, 2014.

[108] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[109] Stefan Panjkovic and Andrea Micheli. Expressive Optimal Temporal Planning via Optimization Modulo Theory. In *Proc. of the AAAI Conference (AAAI)*, volume 37, pages 12095–12102, 2023.

[110] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wellesley, 1984.

[111] Edwin P. D. Pednault. ADL and the State-Transition Model of Action. *Journal of Logic and Computation*, 4(5):467–512, 1994.

[112] Simona Perri and Francesco Scarcello. Advanced Backjumping Techniques for Rule Instantiations. In *APPIA-GULP-PRODE*, pages 238–251, 2003.

[113] Laurent Perron and Vincent Furnon. OR-Tools. https://developers.google.com/optimization. Accessed: 2020-8-11.

[114] Justyna Petke. *Bridging Constraint Satisfaction and Boolean Satisfiability*. Springer, 2015.

[115] Ira Pohl. Heuristic Search viewed as Path Finding in a Graph. *Artificial Intelligence Journal*, 1(3-4):193–204, 1970.

[116] Florian Pommerening, Álvaro Torralba, and Tomáš Balyo. International Planning Competition 2018 – Classical Tracks. https://ipc2018-classical.bitbucket.io/. Accessed: 2020-01-19.

[117] Miquel Ramirez, Nir Lipovetzky, and Christian Muise. Lightweight Automated Planning ToolKiT. http://lapkt.org/. Accessed: 2019-9-20.

[118] Masood Feyzbakhsh Rankooh and Gholamreza Ghassem-Sani. ITSAT: An Efficient SAT-Based Temporal Planner. *Journal of Artificial Intelligence Research*, 53:541–632, 2015.

[119] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks Revisited. In *Proc. of the AAAI Conference (AAAI)*, volume 8, pages 975–982, 2008.

[120] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding Cost-based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[121] Jussi Rintanen. Complexity of Concurrent Temporal Planning. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 7, pages 280–287, 2007.

[122] Jussi Rintanen. Regression for Classical and Nondeterministic Planning. In *Proc. of European Conference in Artificial Intelligence (ECAI)*, pages 568–572, 2008.

[123] Jussi Rintanen. Planning with SAT, Admissible Heuristics and A*. In *Proc. of Int'l Joint Conference on Artificial Intelligence (IJCAI)*, pages 2015–2020, 2011.

[124] Jussi Rintanen. Engineering Efficient Planners with SAT. In *Proc. of European Conference in Artificial Intelligence (ECAI)*, pages 684–689, 2012.

[125] Jussi Rintanen. Planning as Satisfiability: Heuristics. *Artificial Intelligence Journal*, 193:45–86, 2012.

[126] Jussi Rintanen. Madagascar: Scalable planning with SAT. In *Int'l Planning Competition (IPC)*, volume 21, pages 1–5, 2014.

[127] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Parallel Encodings of Classical Planning as Satisfiability. In *European Workshop on Logics in Artificial Intelligence*, pages 307–319, 2004.

[128] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as Satisfiability: Parallel Plans and Algorithms for Plan Search. *Artificial Intelligence Journal*, 170(12-13):1031–1080, 2006.

[129] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[130] Nathan Robinson, Charles Gretton, Duc-Nghia Pham, and Abdul Sattar. A Compact and Efficient SAT Encoding for Planning. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, pages 296–303, 2008.

[131] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[132] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.

[133] Jendrik Seipp and Malte Helmert. Counterexample-guided Cartesian Abstraction Refinement for Classical Planning. *Journal of Artificial Intelligence Research*, 62:535–577, 2018.

[134] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://lab.readthedocs.io/en/stable. Accessed: 2019-10-13.

[135] Bart Selman, Henry Kautz, and David McAllester. Computational Challenges in Propositional Reasoning and Search. In *Proc. of Int'l Joint Conference on Artificial Intelligence (IJCAI)*, pages 50–54, 1997.

[136] Bart Selman, Henry A Kautz, and Bram Cohen. Noise Strategies for Improving Local Search. In *Proc. of the AAAI Conference (AAAI)*, volume 94, pages 337–343, 1994.

[137] Murray Shanahan and Mark Witkowski. Event Calculus Planning Through Satisfiability. *Journal of Logic and Computation*, 14(5):731–745, 2004.

[138] William Shen, Felipe Trevizan, and Sylvie Thiébaux. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, volume 30, pages 574–584, 2020.

[139] JP Marques Silva and Karem A Sakallah. GRASP—A New Search Algorithm for Satisfiability. In *Proc. of International Conference on Computer Aided Design*, pages 220–227. IEEE, 1996.

[140] David E. Smith and Daniel S. Weld. Temporal Planning with Mutual Exclusion Reasoning. In *Proc. of Int'l Joint Conference on Artificial Intelligence (IJCAI)*, volume 99, page 326–333, 1999.

[141] Matthew Streeter and Stephen F. Smith. Using Decision Procedures Efficiently for Optimization. In *Proc. of Int'l Conference on Automated Planning and Scheduling (ICAPS)*, pages 312–319, 2007.

[142] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc Challenge 2008-2013. *AI Magazine*, 35(2):55–60, 2014.

[143] Martin Suda. Property Directed Reachability for Automated Planning. *Journal of Artificial Intelligence Research*, 50:265–319, 2014.

[144] Alvaro Torralba, Vidal Alcázar, Peter Kissmann, and Stefan Edelkamp. cGamer: Constrained Gamer. *Int'l Planning Competition (IPC)*, pages 74–76, 2014.

[145] Alvaro Torralba, Vidal Alcázar, Peter Kissmann, and Stefan Edelkamp. Efficient Symbolic Search for Cost-optimal Planning. *Artificial Intelligence Journal*, 242:52–79, 2017.

[146] Satya G. Vadlamudi, Sandip Aine, and Partha P. Chakrabarti. MAWA*: A Memory-Bounded Anytime Heuristic Search Algorithm. *IEEE Transactions on Systems, Man, and Cybernetics*, 41(3):725–735, 2011.

[147] Alessandro Valentini, Andrea Micheli, and Alessandro Cimatti. Temporal Planning with Intermediate Conditions and Effects . In *Proc. of the AAAI Conference (AAAI)*, volume 34, pages 9975–9982, 2020.

[148] Mauro Vallati, Lukáš Chrpa, and Thomas L Mccluskey. What you always wanted to know about the deterministic part of the international planning competition (IPC) 2014 (but were too afraid to ask). *The Knowledge Engineering Review*, 33:e3, 2018.

[149] Peter van Beek and Xinguang Chen. CPlan: A Constraint Programming Approach to Planning. In *Proc. of the AAAI Conference (AAAI)*, pages 585–590, 1999.

[150] Vincent Vidal. The YAHSP planning system: Forward Heuristic Search with Lookahead Plans Analysis. *Int'l Planning Competition (IPC)*, pages 56–58, 2004.

[151] Vincent Vidal. YAHSP3 and YAHSP3-MT in the 8th International Planning Competition. *Int'l Planning Competition (IPC)*, pages 64–65, 2014.

[152] Vincent Vidal and Héctor Geffner. Branching and pruning: An Optimal Temporal POCL planner based on Constraint Programming. *Artificial Intelligence Journal*, 170:298–335, 2006.

[153] Toby Walsh. SAT v CSP. In *Int'l Conference on Principles and Practice of Constraint Programming (CP)*, pages 441–456, 2000.

[154] Takayuki Yoshizumi, Teruhisa Miura, and Toru Ishida. A* with Partial Expansion for Large Branching Factor Problems. In *Proc. of the AAAI Conference (AAAI)*, pages 923–929, 2000.