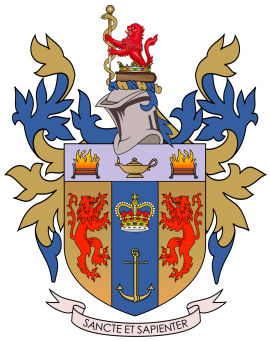
From Pattern Databases to Plan Libraries: Utilising Memory-based Methods for Improving AI Planning Performance

Moraru, Ionut

Awarding institution:
King's College London



From Pattern Databases to Plan Libraries: Utilising Memory-based Methods for Improving AI Planning Performance

Ionut Moraru

Department of Informatics
King's College London

A thesis submitted in partial fulfilment for the degree of
Doctor of Philosophy

May 2023

To my amazing parents,

thank you for all your love and support!

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Ionut Moraru
May 2023

Acknowledgements

I would like to start by expressing my gratitude to my first set supervisors: **Prof. Stefan Edelkamp** and **Dr. Daniele Magazzeni**, for the support and guidance they offered me during my first two years of study. I will always appreciate the opportunity they gave me to do a PhD degree, and start my research and teaching career.

Likewise, I would like to thank the supervisors that took and guided me in the second half of my PhD: **Prof. Simon Parsons**, **Prof. Peter McBurney** and **Dr. Andrew Coles**. They offered me the chance of broadening my AI experience by giving me the chance to work in the field of robotics, and were always supportive during this difficult, but interesting period. I would especially like to thank **Simon**, for showing me how to deliver quality education while always being honest to my students, and to **Peter**, for all the great conversations we had.

None of this would have been possible without the infinite love and support my parents, **Gabriela** and **Sorin-Aurel Moraru**, offered me. I am extremely lucky to have parents and role models like them, who never stop trying to become the best versions of themselves. I would like to extend that gratitude to the rest of my family, to my grandparents **Maria** and **Alexandru** – mamaita and tataita – and to my aunt and uncle.

I am thankful to all the colleagues I had at King's, to **Stefan** and **Francesca** – my London parents, to **Adrian**, **Xavi**, **Moises** and **Akkapon** – the best friends one can have, to **Parisa**, **Senka**, **Diego** and **Gerard** – for the many laughs and tea-breaks, to **Zhuoling**, **Alex**, **Anna**, **Adam**, **Giulio**, **David**, **Leo**, **JLo** and **Alison** – for being great friends and companions, to **Chris** for being the best TA manager in the world, to **Hari** – for being the most helpful person I have ever met.

The past four years have been difficult, but having friends like **Dimitar** and **Vanya** made them a lot easier, and I am thankful for their trust and friendship. I would also like to thank my friends **Andrei**, **Bianca** and **Manu** for always being honest and supportive of my work.

Finally, I would like to thank my amazing girlfriend **Zara**, for all the love, support and attention you have offered me during the last two years. Meeting you during this PhD made it all the more valuable to me!

Abstract

Planning is the field of Artificial Intelligence (AI) tasked with finding a sequence of actions for achieving a goal from an initial description of the environment. In this thesis, we will look into methods of leveraging memory for improving cost-optimal deterministic planning, and leveraging previous plans and executions for agents that operate in dynamic environments.

Cost-optimal deterministic planning systems have been enhanced by using heuristics into their reasoning process, but most of them are either defined by the engineers of the systems or specialised on certain tasks. The automated construction of heuristics is a long-term aim in AI Planning, and Pattern Databases (PDBs) serve as abstraction memory-based heuristics generated prior to the search to enhance computational efforts. Recent work in the automatic generation of symbolic PDBs has established it as one of the most successful approaches for cost-optimal domain-independent planning, being the approach used by the best performing planners in the last two International Planning Competitions (IPC).

We start by proposing two new approaches for combining several patterns into better heuristics, from using Constraint Programing languages (Minizinc) for finding optimal combinations, to sub-optimal but fast approaches using bin-packing algorithms for this. In continuation, we found novel ways of creating competitive patterns, by combining a full deterministic greedy algorithm we call Partial Gamer with bin-packing, which has shown that they complement very well for different types of domains.

We then changed our focus, from theoretical planning tasks to domains for robotic agents. Two major issues of these tasks arise from the stochastic nature of the environment they operate in, and from the high cost of a failure during execution, meaning frequent replanning is required. One way to address this problem is to make use of a pre-defined plan library. Such libraries have been used in Belief-Desire-Intention (BDI) agents for storing behaviour in a computationally efficient manner, however this imposes a limit on the agent's autonomy — it can only do what its plan library allows it to do. AI Task Planning has been integrated into BDI agent

programming languages to improve autonomy by allowing new plans to be created, but with limited success. We present work that combines a plan library with task planning, with results showing that such an approach alleviates the computational burden of synthesising plans, while also observing that the larger a plan library is, and to an extent the more memory used for it, the less time will be spent generating plans.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Over the last decades, Artificial Intelligence (AI) has entered almost all areas of human activity, from social interaction via social media platforms such as Twitter and TikTok, to manufacturing and agriculture. In manufacture related areas, there has been a larger focus on process automatization — using specialised machinery to replicate narrow tasks performed by human personnel [44]. However, most other areas have not been as prepared to integrate AI technologies into their workflow.

This comes due to a variety of reasons, a some of them being:

- The World Wide Web and social media created an *unprecedented scale-up* in how humans shared thoughts and ideas with each other, greatly increasing one's reach with a relatively small cost, and AI techniques accentuated small and localised issues such as though bubbles and groupthink [106]. This has been accentuated due to the narrow approach taken by AI when optimising delivery of information to their users (e.g. viewtime on YouTube, clickrate on Google, etc.). All the while these systems are difficult to be verified and controlled;

- In the field of agriculture, the *dynamic environment* is an issue that computationally overloads most systems and benefits from the adaptability of having a human in control-loop. The main issue a field like agricultural robotics found is that farms are living-breathing environments that are part of nature, and nature is difficult to model or standardize [144]. The weather plays a huge factor in such an environment, and timing of different actions make decision-making difficult for AI systems operating in it;

- Most decision-making processes have been made to be implemented and used by human users, which adapt and reason well while under difficult constraints, while also leveraging previous experience. This made it difficult for AI to be reliably included in fields where they would need to team up and cooperate with humans, as they do not create a sense of familiarity and trust needed for such an integration.

In AI systems, two main approaches have been developed: data-based AI and model-based AI. In data-based AI systems, the focus for identifying the correct answer comes free from any assumptions of how the environment behaves, resulting directly from the data around many examples of correct or wrong answers to the problem. Model-based AI (or Symbolic AI) tries to capture knowledge from the problem and explicitly represent it, such that the system can use it when answering a query or making a decision.

For data-based approaches, such as Machine Learning techniques, the underlying idea is that any possible situation would have been encountered during training, therefore not needing any more information from the current task and solution. For model-based approaches, the system assumes that if the model is correct, there is no need to doubt the reasoning process behind the AI. However, in both cases, we argue that this wasteful approach loses reasoning speed and vital information regarding the nature of the problem.

Current AI techniques perform at human and superhuman level narrow tasks, such as classifying images into different tasks, or playing games like chess, Go and StarCraft, but they treat most problems as a singular event. All reasoning from solving one task is thrown away after execution, without any information being kept for future executions.

1.1 Motivation

In this thesis we set up to broaden the scope of AI Reasoning (i.e. the automated process of taking decisions regarding the environment the system is a part of, for the aim of reaching a goal), by working on techniques that reduce the time spent on reasoning how to achieve goals that have previously encountered. This is done by utilising memory-based solutions for generating heuristics that aid the search for a solution, or searching for previous solutions for the tasks at hand.

We will start by looking at the field of AI Planning, and the memory-based heuristic *Pattern Databases*. This method uses domain abstraction for creating simplified versions of the task

at hand. Proceeding to solve it in an exhaustive style, using the resulted distances from the abstracted problem as a guide for the concrete problem. We will then carry out an analysis of the deterministic 2018 International Planning Competition, where we sent two planners based on Pattern Databases, showing which techniques are currently best suited for tackling domain-indenpendent planning.

Following this technique, we will look into methods integrating AI Planning into the reasoning chain of robotic agents, and utilisation of time-consuming optimal solutions in reasoning for dynamic environments. We will then investigate how previous plans and executions can be taken advantage of when solving new tasks, by taking inspiration from Belief-Desire-Intention agents.

Next, we define the concrete goals of this thesis and the motivation for our work. Finally, we present the structure of the rest of this thesis.

1.2 Thesis Objectives and Contributions

We set ourselves three main objectives that relate to planning and memory-based systems:

The first objective – **O1** – is to investigate the field of cost-optimal classical planning, looking at how best to represent a planning state and methods of combining several heuristics into one best suited for the task given. We will do this by using Pattern Databases (PDBs), an abstraction-based heuristic that has shown great performance across different domains. This combined with a symbolic representation of the planning state, combine into state-of-the-art performance across all modern planning benchmarks. Early versions of the techniques revolving around PDBs were submitted as independent planners to the Cost-Optimal track from the 2018 deterministic International Planning Competition (IPC), in which it proved as the most powerful novel search approach available at that time.

The second objective – **O2** – is to do an analysis of the Cost-Optimal track from the 2018 Deterministic International Planning Competition (IPC), where as stated previously, our two planners finished inside the top four, both within 2% distance from the winner. The IPC is an important event for the planning community, as it is a fair evaluation of the field as a whole, and a good analysis of it will lead to advances and important information regarding what are the new best approaches to solving domain independent planning. We identify this competition as a valuable source of information and practical solutions to planning as a whole. This, together

with the basis of PDBs and Pattern Selection, directed us to shift our focus into AI planning for robotics and research how to best use deterministic planning in dynamic environments.

Lastly, the third objective – **O3** – directly follows the work presented in the first two parts, and is aiming at developing a method that could leverage past solutions in executions for aiding agents that need to reason in environments with uncertainty.

The main contributions of this thesis are as follows:

- **C1 –** Improve cost-optimal planning with Pattern Databases, as this method takes most advantage of the memory gains from the hardware of the planning system;
- **C2 –** Evaluate different Pattern Selection systems, with the aim of finding the best combination of solutions, while simplifying the creation process;
- **C3 –** Define portofolio-planning, which was the most interesting revelation from the cost-optimal track of the deterministic IPC. The good performance from the competition is due to leveraging knowledge from previous planners' executions on existing planning problems;
- **C4 –** Create a better method to evaluate domain-independence, which is one of the aims for AI planning systems. Current methods show biases towards older planning problems, while also minimising the evaluation to a singular number;
- **C5 –** Reuse plans via plan libraries for improving long-term execution for systems that operate in dynamic environments;
- **C6 –** Investigate and create different plan selection techniques, for agents that operate with plan libraries.

1.3 Thesis Structure

This thesis has the following structure:

1. **Background and Related Works –** Chapter two will offer all the background and prerequisite information necessary for understanding the work in this thesis. We will also have a review of works related to the ones we investigate and propose in this thesis.

2. **Bin-Packing and Greedy Selection for PDB Creation –** In Chapter three, our aim will be to define the process of automatically generating abstraction-based heuristics. We will follow this by introducing the Pattern Selection problem, and we will propose three new methods for solving it. We will evaluate each one across all deterministic benchmarks, discussing how the approaches affect the results. This chapter will investigate the first objective – **O1** – and result in contributions **C1** and **C2**.

3. **Cost-Optimal Track of the Deterministic IPC18 –** This chapter will include an analysis of the Cost-Optimal track from the 2018 Deterministic International Planning Competition, offering new definitions for portofolio-planning and offering conclusions and metrics on how to best solve cost-optimal planning. Objective **O2** will be the aim of this chapter, with a contributions **C3** and **C4** resulting from it.

4. **AI Planning with Robotics by using Plan Libraries –** Chapter five will include a look into the field of planning under uncertainty for robotic agents, and how Plan Libraries, an idea inspired by Belief-Desire-Intention agents, can be used to make deterministic planning a fast and robust solution. The final objective – **O3** – is the focus of this chapter, resulting in the final two contributions, **C5** and **C6**.

5. **Conclusion and Future Work –** The final chapter will include a review of our contributions, and a look back at how the objectives of this thesis have been met. We will then conclude with a look at what future research problems have come out from our results.

1.4 Publications

This thesis includes research and results published in the following peer-reviewed papers at conferences, workshops and two planner abstracts from an international competiton taking place at a conference:

1. Moraru, I., Edelkamp, S., Franco, S., & Martinez, M. (2019). Simplifying automated pattern selection for planning with symbolic pattern databases. In Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz) (pp. 249-263). Springer, Cham. [102]

2. Moraru, I., Canal, G., & Parsons, S. (2021). Using Plan Libraries for Improved Plan Execution. In UKRAS21 Conference:“Robotics at home” Proceedings. (pp. 51-52). [100]

3. Munoz, M. M., Moraru, I., & Edelkamp, S. (2018). Automated Pattern Selection using MiniZinc. In International Conference on Principles and Practice of Constraint Programming: Workshop of Constraints and AI Planning. [104]

4. Moraru, I., Edelkamp, S., Martinez, M., & Franco, S. (2019). Simplifying Automated Pattern Selection for Planning with Symbolic Pattern Databases. In ICAPS 2019 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP), (pp. 46-54). [103]

5. Moraru, I., & Edelkamp, S. (2019). Benchmarks Old and New: How to compare domain independence for cost-optimal classical planning. In ICAPS 2019 Workshop on the International Planning Competition (WIPC) (pp. 36-39). [101]

6. Edelkamp, S., & Moraru, I. (2019). Cost-Optimal Planning in the IPC 2018: Symbolic Search and Planning Pattern Databases vs. Portfolio Planning. In ICAPS 2019 Workshop on the International Planning Competition (WIPC), (pp. 15-21). [33]

7. Martinez, M., Moraru, I., Edelkamp, S., & Franco, S. (2018). Planning-PDBs planner in the IPC 2018. IPC-9 planner abstracts, (pp. 63-66). [93]

8. Franco, S., Lelis, L. H., Barley, M., Edelkamp, S., Martines, M., & Moraru, I. (2018). The complementary2 planner in the IPC 2018. IPC-9 planner abstracts, (pp 28-31). [42]

Chapter 2

Background and Related Works

In this chapter, we will describe the background needed for achieving the objectives set in the first chapter. We will start by formally defining the field of Artificial Intelligence Planning, focusing on Classical Planning and formal descriptions of planning tasks. Furthermore, we will describe the current state-of-the-art in Heuristic Search techniques that are based on abstraction and are used for planning cost-optimal solutions, focusing mostly on *Pattern Databases* (PDBs).

The second half of this chapter will include a description of AI Planning applied to Robotics, with an introduction to the Robot Operating System (ROS) and the ROSPlan framework. We will conclude this chapter by going over the works relating to the objectives and contributions of this thesis.

2.1 Artificial Intelligence

Artificial Intelligence (AI) is a field with a surprisingly long history, arguably dating back almost a millennium, with the first search algorithm being developed by the great philosopher and polymath Ibn Sina (known in western culture by his Latinized name, Avicenna) [68]. Known mostly for his work on medicine, metaphysics and ethics, he was also a logician, with new translations of his work revealing a proof search algorithm for syllogisms.

It can be considered telling that the field that was born in the middle of the Islamic Golden Age, a time that greatly evolved human thought, has been uncovered and brought to the forefront in the late 20th/early 21st century, as both eras led to unprecedented growth and prosperity.

However, both periods had to navigate the huge disruptions in their societies and divisions in their populations.

AI in the 1950s and 1960s was described by Marvin Minsky [99] as being split into five major areas: *Search*, *Pattern Recognition*, *Learning*, *Planning* and *Induction*. During this period, AI was envisioned as a symbolic-based general problem and theorem solver. This is nowadays referred to as *Good Old Fashioned Artificial Intelligence* (GOFAI) [58], in contrast to the more data-driven approaches that are now popular. Most modern AI systems are now characterised and described based on the *environment* present and the *agents* affecting it [124].

This thesis focuses on the area of AI that Minsky referred to as *Planning*, which is the discipline tasked with making agents reason about sequential decision-making problems – which actions they should take so that they can achieve a desired state, while taking into account their intended and unintended consequences [94, 119].

From an engineering point of view, we define *Planning* as a system that, when tasked with a complex problem, will produce a sequence of actions that can achieve the intended goal by combining work from several fields, such as knowledge representation, heuristic search, inferences, model abstraction, monotonic/non-monotonic logic, etc.

2.2 AI Planning

AI Planning is described by Hector Geffner [49] as following the solver paradigm. Similar to other model-based AI problems, such as Constraint Satisfaction, SAT, Bayesian Networks, Markov Decision Processes, etc..., solvers use well-defined and sound mathematical description languages [57] in which the task is modeled. For AI Planning, this specialised software is called a *planner*.

2.2.1 Planners

Planners use, as an input, models describing the agents and the environment they act in (i.e. planning task). Most of them currently use validator software to check the correctness of the models, most commonly VAL [74]. Following successful validation, they will then parse the model into a search space or logical reasoning problem, for the software to efficiently tackle the task and output a solution, which is called a *plan*. In the case that no plan is found, it will

output an error, where some planners additionally implement a component that tries to explain why it stopped working.

A generic planning task can be characterised by four elements: a set of variables, a set of operators (i.e. actions), an initial state and a goal state. The variables describe the elements of the environment that are important for completing the task and the operators define how the variables change.

Planning tasks use states to describe the situation (i.e. an assignment of values to the variables describing the task). The initial state is a complete assignment of values to the variables (i.e. complete state), which informs the solver of how the environment is positioned at the beginning of execution. The goal state is a partial assignment of values to the variables (i.e. partial state), describing how the environment should look at the end of executing a plan received from the solver. All the variables not present in the goal state can have any assignment possible in their set.

2.2.2 Planning Task Example

In this thesis, we will be using the office domain example, a problem that was designed based on the office where we worked. In this task, there is one robot called *Freddie*, and a can of water. Freddie's task is to grab the can and place it into the appropriate location. There are three waypoints in the domain, `wp1`, `wp2` and `wp3`, each one being connected to the other. This environment is described in Figure 2.1.

Freddie is able to navigate between the locations. He can also grasp or place objects on surfaces he scans in front of them.

2.2.3 Planning Environments

Planners can be, and have already been, used in an array of different use cases, such as the Mars rover [36], autonomous underwater vehicles in deep sea research [11], the movement of conveyors in a greenhouse [65], generating new stories and narratives [123], and computer bridges [133]. All of these use cases take place in different environments, which pose different issues to the designers of the model.

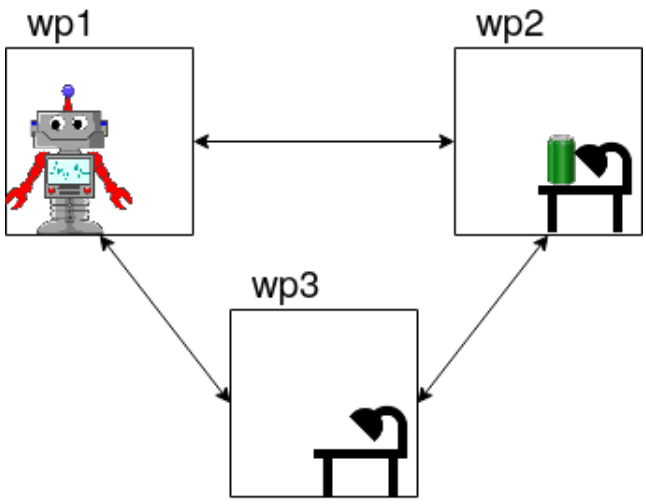


Fig. 2.1 A simplified version of the Office domain, one in which a robot assistant is tasked with preparing the workspace for its users, from cleaning meetings rooms to fetching objects for employees.

Planners are designed to accept specific models as inputs, with most being described in the Planning Domain Description Language (PDDL) (more on this in the next sections). Models differ through the assumptions they make based on how the environment and the agents inside it behave.

These assumptions are split into:

- **Static/Dynamic -** If the environment changes only as a result of an action performed by the agent, the environment is *static*. Otherwise, it is *dynamic*;
- **Discrete/Continuous -** Environments that consist of variables which can be described in a finite number of values are *discrete* (e.g. Chess), while any other is *continuous* (e.g. football);
- **Deterministic/Probabilistic -** If the current state of the environment is determined fully by the previous state and the operator applied by an agent in it, then the environment is called *deterministic*. In other words, if the effects of actions are not always the same, then the agent acts in a *probabilistic* environment;
- **Single agent/Multiagent -** Problems where there are agents that have goals, but are not included in the goal state given to the planner are called *multiagent* problems, as the planner needs to reason about the behaviour of the other agents without having direct access to their reasoning process. Otherwise, it is a *single agent* problem/environment;

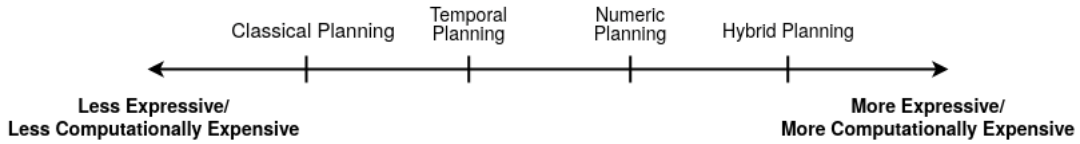


Fig. 2.2 Expressivity of planning formalism correlates directly with the computational cost of the task. The more expressive a formalism is, the more intractable the computational task becomes. Probabilistic planning is excluded from this figure as it can be combined with any of the others, increasing the computational effort of each one.

- **Fully/Partially observable -** In the case that an agent's sensors have access to the complete state of the environment, the environment is called *fully observable* as the agent can reason about all aspects that are in the environment when choosing an action. *Partially observable* environments are more common in applications taking place in the real-world, as sensors are limited in range and can return noisy readings of the surrounding environment;

- **Sequential/Episodic -** *Episodic* environments are characteristic to AI assistants / classification tasks, as they receive a reading emitted by the environment and act upon it, with further actions not depending on the prior actions. In contrast, *sequential* problems are defined by the fact that any current decision will have an effect on future decisions – *short-term actions can have long-term consequences* [124]. AI planning is used most of the time in sequential environments;

- **Temporal/Non-temporal -** If operators need to be differentiated due to the different lengths or specific deadlines certain goals need to be achieved by, then the environment is *temporal*. These problems are specific, but not exclusive, to scheduling problems. *Non-temporal* environments relax this problem, ignoring that actions have different durations.

2.2.4 Planning Models

Several types of AI Planning models have been created, depending on the specific planning tasks that they intend to solve. This was because each problem presented different assumptions suited for their environment, and considered important elements and constraints that other models ignored or omitted.

This was done as the problems had different assumptions that fit the environment better, as well as different important facts and constraints that other models ignored/did not have. Some of the most used single-agent, AI Planning models which are based on PDDL can be categorised[1] as:

- **"Classical" Planning -** The name "classical" was given due to it being the initial version. Classical planners provide the simplest version of planning and representation of its state, assuming its environment to be deterministic, fully-observable, with a discrete state-space and ignores the temporal aspects of its problems. They are most of the time characterised by being single agent;
- **Numeric Planning -** Adds continuous variables to planning problems, allowing the solution of tasks that need to model resource management, specific locations in the environment, etc. [41];
- **Temporal Planning -** It introduces durations to actions, allowing planners to tackle scheduling problems. It also adds events and concurrency to help create a more expressive language [41];
- **Hybrid Planning -** The most expressive version of planning, it combines the previous versions of planning, plus adding continuous processes [40]. Its name comes as it is a combination of both numerical and temporal planning;
- **Probabilistic Planning -** This version of planning can be combined with each of the previous (but most of the time with classical planning), by introducing uncertainty into the effects of each action [147, 126].

From a practical standpoint major difference between all the planning models is the computational complexity of solving each one. As expressed in Figure 2.2, the more *expressive* and feature heavy a model is – and through extension the state needed to represent it completely – the more computational effort it will take for a planner to solve it. Each addition to the description of the state-space, such as duration, cost or consumption, will result result into more computational *time* spent searching for possible solutions to the planning task. As planning is at least P-Space Complete, increasing the size of the state will exponentially increase the time it will take to find a solution, resulting in only shorter plans being achievable for more complex models.

[1] This is by no means an exhaustive topological list of AI Planning types, as it ignores vasts areas of Planning such as Hierarchical Task Network (HTN) planning, but it is sufficient for following the contents of this thesis.

2.2.5 Classical Planning

Classical planning is the simplest and least expressive description of a planning model, with representative problems in the fields of logistics and path-finding in a directed labeled graph, where the nodes of the graph are the possible states of the problem and edges represent the transition that each action makes.

Within classical planning, there are multiple problems that differ in the amount of resources they receive or in their objective. The most basic objective is *plan existence*, where a planner must decide if a plan is possible, given a planning task. Another problem is *satisficing planning*, which aims to combine speed with plan quality - finding as good of a plan as possible (i.e. cost of all actions should be minimised) in a short amount of time.

Cost-optimal planning is the form of planning that is tasked with finding the best plan with regards to the total cost of its actions. Another way in which this type of planning differs from other forms of planning tasks, is that while others can return multiple solutions, typically in increasing quality as they extend their search in the problem space, cost-optimal planning returns only one[2] solution that is guaranteed to be optimal in quality.

While classical planning problems have been shown to be at least `PSPACE-Complete` [3], cost-optimal planning, the most computationally intensive type of planning task, exceeds that complexity barrier most of the time.

As seen in Figure 2.2, all other formalisms of planning are more computationally intensive, and the state-space explosion due to more variables makes fairly small planning tasks impossible for planners to solve in practice. With this knowledge, classical planning is the only type of planning where offering solutions that are cost-optimal is viable and has been a main field of research.

Cost-optimal planning has been used to guide the research field to find new solutions and techniques that can then be applied efficiently into suboptimal scenarios. This has also been due to optimal solutions being used to quantify how good a satisficing solution is, resulting in a quality measure for all other planning types.

As the area of classical planning has had a lot of research poured in, other planning types have been using it for finding solutions to their problems. One example is the area

[2]For some planning tasks, there could be multiple solutions that have the same optimal cost, and in practice they would be of interest to compare. However, from a theoretical standpoint, they would all be the same and as such that planner will terminate after finding the first one.

of probabilistic planning, where many of the best planners in the probabilistic track of the International Planning Competition have been *determinizing* the probabilistic task and sending it to classical planners to find solutions for them. This approach has shown great success, with FF-Replan [146] winning in 2004 and still performing best on the benchmarks from 2007.

2.2.6 Fast Downward Planning Framework

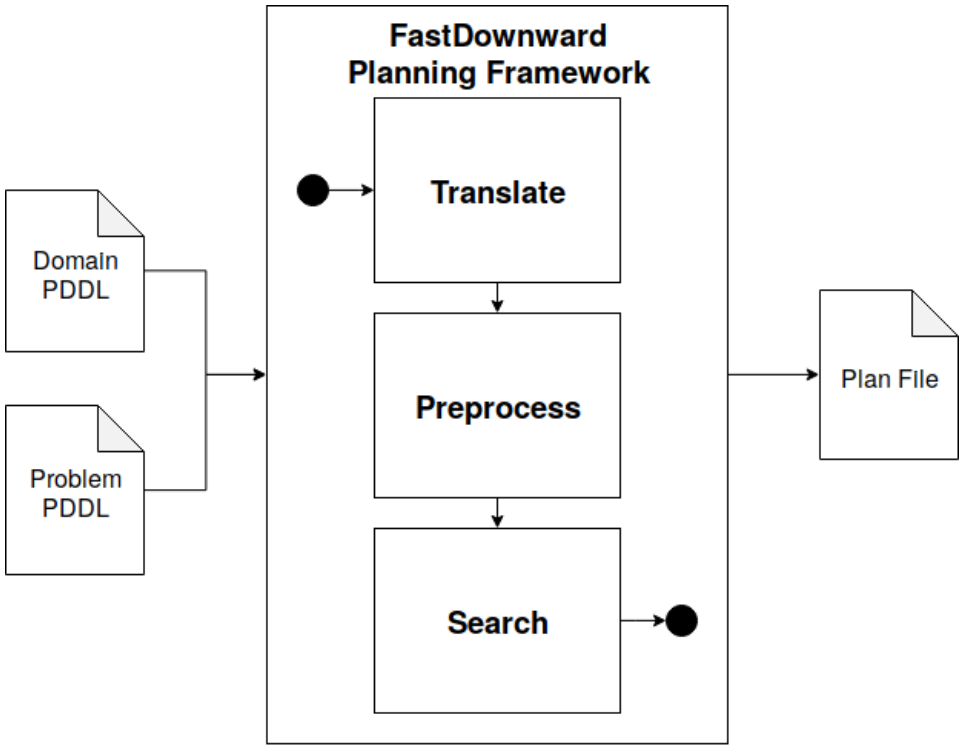


Fig. 2.3 High level representation of a planner based on the Fast Downward family of planners [61]. More recent versions of it have combined the translate and preprocess components into one, as most of the planning community have only been extending the search component.

One of the most influential planners has been Fast Downward (FD), by Malte Helmert [61]. Figure 2.3 illustrates its modular approach, where it is split into three components: *translate*, *preprocess* and *search*.

- **Translate -** tasked with parsing the domain/problem PDDL input files into a multi-valued planning tasks, similar to that given by the SAS$^+$ representation, while also doing invariant synthesis and grounding all the variables;

- **Preprocess -** based on the files received from the translation component, it will create data structures that will aid when trying to solve the planning task, such as *domain transition graphs*, *causal graphs* and *successor generators*.
- **Search -** the component that implements the search algorithms used for finding a plan. The initial implementation included three different search components, *greedy best-first search*, *multi-heuristic best-first search*, and *focused iterative-broadening search*. Since then, it has been extended to include most search algorithms and heuristics applicable for planning [66].

2.3 Formal Descriptions of Classical Planning Tasks

To specify planning tasks to a solver system, there needs to be a problem specification language in which a problem can be expressed. Over the years, many formalisms have been developed, each initially tackling different types of problems.

2.3.1 STRIPS

The first and most influential formal definition of a planning task is STRIPS (STanford Research Institute Problem Solver) [39]. Its main objective was to create a formal description which could be used to describe problems for *Shakey the Robot* [108], such as route finding and moving items. Shakey was a major breakthrough, as it was the first mobile robot that had good and reliable sensing abilities, allowing him to reason its environment and how they could affect it.

STRIPS can be defined as follows:

Definition 1 (Propositional STRIPS Planning Task) *The tuple* $\langle \mathscr{A}, \mathscr{O}, \mathscr{I}, \mathscr{G} \rangle$ *defines the STRIPS planning task* $\mathscr{P}$*, where:*

- $\mathscr{A}$ *is a finite set of ground atomic formulas, called the* `condition`*;*
- $\mathscr{O}$ *is a finite set of* `operators`*, where each operator consists of:*
  - `Pre-conditions`*, which are satisfiable conjunctions of* `conditions`*, either positive* (o^+) *or negative* (o^-)*;*

  - *`Post-conditions`, which are satisfiable conjunctions of `effects` (changes to the `conditions` once the operator is applied), either positive (o_+) or negative (o_-), also known as the add and delete lists.*

- *$\mathscr{I}$ is the `initial state`;*
- *$\mathscr{G}$ represents a set of goals, which is defined as a satisfiable conjunction of positive and negative `conditions`.*

A solution to a STRIPS planning task is called a *plan*, which would be a sequence of ordered actions that would allow the agent to reach its goal state, starting from an initial state.

Definition 2 (Plan) *A sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ is said to be a plan for a STRIPS planning task, if and only if applying the actions from π on the initial state $\mathscr{I}$, will reach a state where $\mathscr{G}$ is true.*

Following the success of STRIPS, research in the area of action planning grew and resulted in several other formal descriptions being created. ADL (Action Description Language) [112] was based on the state-transition model of actions, and it combined the computational benefits brought by STRIPS and the power of calculus.

2.3.2 SAS and SAS$^+$

Simplified Action Structures (*SAS* or *SAS*$^+$) [125], gets its name from the constraints placed on the action structure. The formalisms differ from STRIPS by using multi-values state variables in place of propositional atoms and splitting operators pre-conditions into *pre-conditions* and *prevail-conditions*, where the difference is by the need to specify which values will be true, even if they will be unchanged after the application of the operator.

Definition 3 (SAS$^+$ Planning Task) *Is a tuple $\mathscr{P} = \langle \mathscr{V}, \mathscr{O}, s_0, s_* \rangle$, where:*

- *$\mathscr{V} = \{v_1, \ldots, v_n\}$ is the set of finite-domain variables. Each variable $v \in \mathscr{V}$ has an associated domain $\mathscr{D}_v$, which implicitly defines the extended domain $\mathscr{D}_v^+ = \mathscr{D} \bigcup \boldsymbol{u}$, where $\boldsymbol{u}$ denotes the undefined value. The **total state space** $\mathscr{S}_{\mathscr{V}} = \mathscr{D}_{v_1} \times \ldots \times \mathscr{D}_{v_n}$ and the **partial state space** is implicitly defined as $\mathscr{S}_{\mathscr{V}}^+ = \mathscr{D}_{v_1}^+ \times \ldots \times \mathscr{D}_{v_n}^+$*

- $\mathcal{O}$ *is a set of operators which consist of pre-, post- and prevail-conditions, where* $\texttt{pre}, \texttt{post}, \texttt{prv} \in \mathcal{S}^+_{\mathcal{V}}$. *Each* $\langle \texttt{pre}, \texttt{post}, \texttt{prv} \rangle \in \mathcal{O}$ *is constrained as follows:*
  - *for all* $v \in \mathcal{V}$, *if* $\texttt{pre}[v] \neq \texttt{u}$, *then* $\texttt{pre}[v] \neq \texttt{post}[v] \neq \texttt{u}$;
  - *for all* $v \in \mathcal{V}$, *if* $\texttt{post}[v] = \texttt{u}$ *or* $\texttt{prv} = \texttt{u}$
- s_0 *and* s_* *are* ***states***. *A (complete) state* $s = \langle a_1, \ldots, a_n \rangle \in \mathcal{S}$ *assigns a value* a_i *to every* $v_i \in \mathcal{V}$, *with* a_i *in a finite domain* D_i, $i = 1, \ldots, n$. *We have* $s_0 \in \mathcal{S}_{\mathcal{V}}$ *and* $s_* \in \mathcal{S}^+_{\mathcal{V}}$.

For the office example that we defined at the beginning of the chapter, we can use two variables to define the planning task: *robot-location* and *can-location*. The domain of the two variables is $\mathcal{D}_{robot\_location} = \{wp1, wp2, wp3\}$ and $\mathcal{D}_{can\_location} = \{wp1, wp2, wp3, Freddie\}$. We can then state that the can is in Freddie's gripper by having the tuple $\langle can\_location, Freddie \rangle$, which we call an atom.

Definition 4 (Atom) *Let* $\mathcal{V} = \{v_1, \ldots, v_n\}$ *is the set of finite-domain variables. We can define an* atom *of a variable x as the tuple* $\langle \mathcal{V}_x, v \rangle$, *where* $\mathcal{V}_x$ *in* $\mathcal{V}$ *and v in* $\mathcal{V}_x$.

We can then use this to refer to a complete state as having atoms for each one of the variables in $\mathcal{V}$ of the planning task. In our example (see Section 2.2.1), where the initial state s_0 is shown in Figure 2.1, we can refer to the initial state the set defined by the two atoms:

$$\{\langle robot\_location, wp1 \rangle, \langle can\_location, wp2 \rangle\}$$

For the goal condition s_* to be consistent with the state, it is defined as the set consisting of:

$$\{\langle robot\_location, wp2 \rangle, \langle can\_location, wp3 \rangle\}$$

Operators in the Office domain are defined as:

$$\mathcal{O} = \{goto - locX - locY, grasp\_obj, place\_obj\}$$

in which grasping and placing can only be applied in *wp2* and *wp3* (the only locations with desks where an object can be placed), while Freddie can apply *goto-locX-locY* between each location. Each operator will have a set $\langle pre, post, prv \rangle$. For example, the *grasp* operator cannot be applied if the two variables do not have the same assignment and will result in the location of

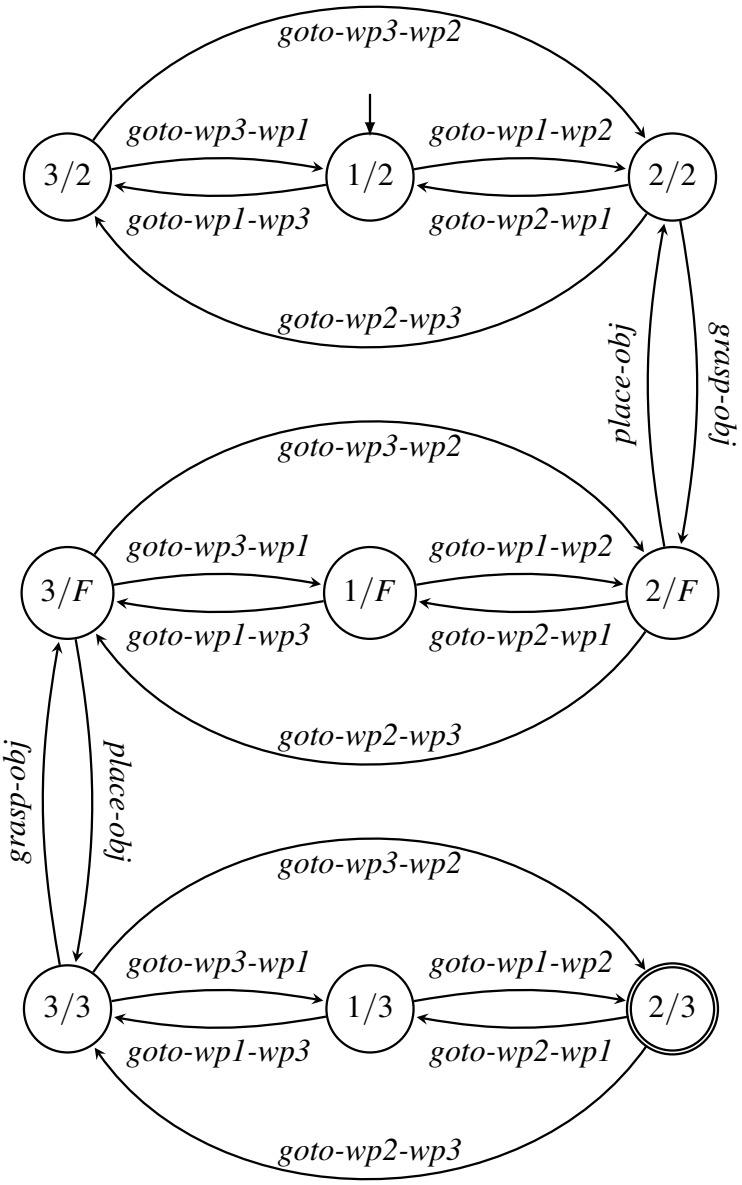


Fig. 2.4 Transition state for the office domain example. The nodes have been simplified to X/Y format, representing a state $\{\langle robot\_location, X\rangle, \langle can\_location, Y\rangle\}$

the can being Freddie, with Freddie's location continuing to be the same as before the operator was applied (i.e. prevailing).

The transition state for the office domain can be seen in Figure 2.4.

In the 1990s, encouraged by the promising results of heuristics approaches based on domain homomorphism and state abstraction in domains such as the Rubiks Cube [84] and the 15-Puzzle [20], Hernádvölgyi and Holte [72] created the Production System Vector Notation (PSVN). It innovated by representing states as vectors of labels that are of fixed length. It was designed for permutation games and for generating abstraction-based heuristics.

2.3.3 PDDL

Finally, by combining concepts from propositional STRIPS and ADL, the Planning Domain Description Language (PDDL) [95, 41] was developed in the late 1990s. PDDL was created as a standardized language for defining planning tasks, used for all the editions of the International Planning Competition (IPC), with the first one taking place in 1998. PDDL and the IPC are tightly connected and have benefited each other over the years, with newer versions of PDDL usually being released prior to one of the competitions (e.g. PDDL2.2 was released in conjunction with the Classical Part of the 4th IPC, 2004 [30]), leading to the research community exploring the capabilities of the modeling language in a quick and easy manner.

PDDL has been extended several times since its inception [41, 30, 51, 40], adding capabilities to model temporal actions, continuous and probabilistic effects, agent preferences or derived predicates.

As all editions of the IPC have been using PDDL to define their sets of benchmarks, it became the modelling language with the largest amount of readily available planning tasks for researchers in the field. This has resulted in it becoming the de facto modeling language in the AI planning community. PDDL works by separating the planning task into a domain and problem file. This change has made planning tasks more modular, as more problems (where the initial state and the goal state are defined) can use the same domain (in which the types of variables and actions are described). This makes it easier to create problem/domain combinations of increasing difficulty [136] for testing the limits and evaluating the strengths and weaknesses of different planners.

When looking for cost-optimal solutions, planners using informed search algorithm, such as A* [53] with domain-independent heuristics, have dominated IPCs over the previous editions [17, 137]. The modular approach developed in FD has made it easy for others to extend and develop heuristics in it. As a result, most planners in the classical tracks of the IPC have been different implementations of FD.

2.4 Symbolic Planning

Most state-based planners use methods that operate on explicit states, where each assignment of different values to the variable in the planning task is a different state. Representation can be beneficial for certain problems, but it has shown to be prone to scaling issues when working in domains that have many predicates, leading to a state-space explosion. This is due to the exponential growth of the state-space depending on the number of variables.

Symbolic planning differs from this by representing a state as a Boolean function, leading to applying operators on sets of states, instead of one by one changes. The first big advantage of such an approach is that it is searching for a solution that uses less memory, as states are compressed based on their logical representation which now grows linearly, instead of exponentially.

2.4.1 Binary Decision Diagrams

The best suited data structures for symbolic planning are *binary decision diagrams* (BDD), which were created based on the visual representation of a logical function.

Definition 5 (Binary Decision Diagrams) *A binary decision diagram is a directed acyclical graph, with a single root node and two end nodes, 0 or 1 (i.e. True and False). Each node is labelled by the variable it represents, and has two edges, labelled 0 or 1.*

An example of how the reduction of the BDD affects a state is shown in Figure 2.5.

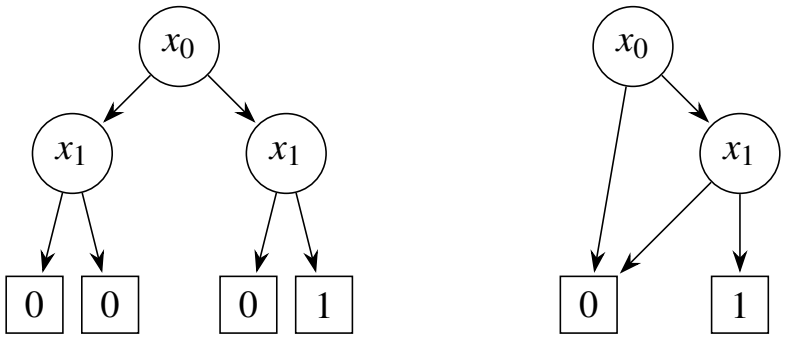


Fig. 2.5 On the right, a Reduced and Ordered Binary Decision Diagram (ROBDD), that is created based on the unreduced BDD from the left [35].

Reduced and ordered BDDs are considered to be best suited for representing planning problems.

Definition 6 (Reduced and Ordered BDD) *A BDD is considered reduced and ordered when:*

1. *Each path to a fixed ordering of variables is preserved;*
2. *Nodes with the same successors are removed;*
3. *Isomorphic sub-BDDs are merged.*

Symbolic search algorithms vary from blind search algorithms like unidirectional or bidirectional uniform-cost search to heuristic guided search like A*. For the symbolic implementation of A* also known as BDDA*, there are also two lists of states, *open* (the generated but not yet explored states) and *closed* list (the states that have already been explored), which are represented as BDDs.

BDDA* differs from explicit A* by grouping sets of states with the same heuristic values and distances from the initial state into g,h-buckets, which can then all be explored at the same time. A heuristic value h for set of states S can be found with the conjunction $S \wedge h_I$, which corresponds to the subset of states that have an h-value equal to i.

2.5 Planning as Heuristic Search

Heuristics are defined by Edelkamp in [35] as a method of evaluating the quality of a state by estimating the remaining distance from a state (i.e. node in a graph) to the goal. As planning is hard, they are relaxations of the problem's constraints that solve a relaxed version for the planning tasks exactly. By using this information, we can do an *informed* search through the state space and access one state that is *more promising* than the rest.

Definition 7 (Heuristic) *A heuristic is a mapping h of the set of states in $\mathscr{P}$ to positive reals $R_{\geq 0}$.*

Algorithm 1 A* Search Algorithm

Require: Graph G, initial node *initialNode*, weight function w, heuristic h, successor function *Expand* and the goal function *Goal*

function A*(G, $initialNode$, h, $Expand$, $Goal$):
  $OpenList \leftarrow initialNode$
  $ClosedList \leftarrow \emptyset$
  $f(initialNode) \leftarrow h(initialNode)$
  while $OpenList \neq \emptyset$ **do**
    $currentNode \leftarrow$ element from $OpenList$ with minimum $f(OpenList)$)
    Insert $currentNode$ into $ClosedList$
    Remove $currentNode$ from $OpenList$
    if $Goal(currentNode)$ **then**
      return $Path(currentNode)$
    else
      $Succ(currentNode) \leftarrow Expand(CurrentNode)$
      for each $succNode$ in $Succ(currentNode)$ **do**
        $Improve(currentNode, succNode)$
      end for
    end if
  end while
  return $\emptyset$
end function

Require: $currentNode$, $succNode$, $f(succNode)$, $OpenList$, and $ClosedList$

function IMPROVE:
  if $succNode$ in $OpenList$ **then**
    if $(g(currentNode)) + w(currentNode, succNode) < g(succNode))$ **then**
      $parent(succNode) \leftarrow currentNode$
      $f(succNode) \leftarrow g(currentNode) + w(currentNode, succNode) + h(succNode)$
    end if
  else if $succNode$ in $ClosedList$ **then**
    if $(g(u) + w(u,v) < g(v))$ **then**
      $parent(succNode) \leftarrow currentNode$
      $f(succNode) \leftarrow g(currentNode) + w(currentNode, succNode) + h(succNode)$
      Remove $succNode$ from $ClosedList$
      Insert $succNode$ into $OpenList$ with $f(succNode)$
    end if
  else
    $parent(succNode) \leftarrow currentNode$
    Initialize $f(succNode) \leftarrow g(current) + w(currentNode, succNode) + h(succNode)$
    Add $succNode$ into $OpenList$ with $f(succNode)$
  end if
end function

2.5.1 A* Search

Informed search algorithms, such as A* which is described in Algorithm 1 or Greedy Best-First Search (GBFS), operate by keeping in memory two lists of planning states called *open* and *closed* lists. In the *open* list, the algorithm maintains all the nodes that were generated in the planning graph (i.e. search tree), while the *closed* list keeps all the nodes which have already extended.

Nodes from the *open list* are selected for expansion depending on their f-values, which for A* is defined as the sum of all cost / weight from the initial node to the node in the list (g-value + w-value), plus the heuristic value (h-value). GBFS differs by selecting nodes only based on their heuristic value, resulting in suboptimal solutions given in a short amount of time.

Heuristics can have four properties: *domain-independence*, *admissibility*, *consistency* and *aditivity*. Heuristics are domain-independent if they were designed to solve any planning task that can be specified in the planning model used. If they were created to solve a specific task, they are called domain-specific heuristics.

A property of heuristics that is essential for finding optimal plans is *admissibility*, because this property guarantees that the solution returned will be optimal when using search algorithms such as A* [53].

Definition 8 (Admissibility) *A* heuristic *is called* admissible*, if $h(s)$ is always a lower bound (i.e. underestimate) of the cost of all goal-reaching plans starting from state s.*

If heuristics are *consistent*, then during the search of the state in the problem, no node will be expanded more than once (i.e. added to the open list several times). This means that once a specific node has been explored and no solution was found, then after backtracking to a previous node in the search tree, the algorithm will not explore that node via a different path. This helps improve the efficiency and is important in finding optimal plans when given a time limit.

Definition 9 (Consistency) *A heuristic is* consistent *if for all operators o from s to s' we have $h(s') - h(s) + c(o) \geq 0$.*

Holte et al. [71] showed that multiple abstraction-based heuristics can be generated for the same planning task and added up, leading most of the time to better performing heuristics.

When the resulting heuristic value is still admissible, then we call the initial heuristics *additive*. We will come back to this property in a later chapter.

Definition 10 (Additivity) *Two heuristics h_1 and h_2 are* additive*, if h defined by $h(s) = h_1(s) + h_2(s)$ for all $s \in \mathscr{S}$, is admissible.*

2.5.2 Heuristic Classification

The field of heuristic search used for finding plans can be split into five different families of heuristics [63]:

- **Delete relaxation -** estimate the cost to the goal by not taking into consideration the negative effects of actions, i.e. considering that actions can add more facts to be true, but do not delete any facts that have already been achieved. E.g. max heuristic [6], h^+ heuristic [69] and the landmark-cut heuristic [63];
- **Abstraction -** estimate the cost to the goal by applying an abstraction on the state-space and lead to a smaller space and simpler planning task to solve. For state-space $\mathscr{S}$ and an abstraction function α, the heuristic function $h^\alpha(s)$ for state s is the cost of the cheapest path from $\alpha(s)$ to a goal state in $\mathscr{S}^\alpha$. E.g. pattern databases (PDBs) [21, 29], symbolic PDBs [26], Cartesian abstractions [127] and merge-and-shrink [64];
- **Critical Paths -** estimate the cost of reaching a set of atoms as the cost of the most costly subset of size m. More formally, a set of sub-goals is considered reachable only when all size-m subsets are reachable ($m \in \mathbf{N}_1$ is a parameter). E.g. h^m heuristic family [50], additive h^m [54] and additive-disjunctive heuristic graphs [18];
- **Landmarks -** are facts that have to be true at some point in every plan in order for it to be a successful plan. More formally, an action set A is a landmark if all plans include an action from it. For this family of heuristics, compute a set of landmarks and use it to derive a cost estimate. E.g. cost-partitioned landmarks [78] and landmark-cut heuristic [63];
- **Network Flows -** this type of heuristic takes into consideration the fact that in every plan, the number of times each fact is *produced* and the number of times it is *consumed* must be correlated. This is solved by mapping the planning problem into a Linear programming task. E.g. flow heuristic [140], state equation heuristic [5] and one heuristic that tackles the relationship between flow heuristics, abstraction and cost partitioning [115].

2.5.3 Abstraction Heuristics

Marvin Minsky [99] was the first to define and use abstraction transformations for aiding an algorithm when searching for a solution in the problems space. He referred to the need of finding useful problem simplifications so that an *incomplete analysis* of a task will result in reliable results, by making note of the intractability of any *worthy* problem and not wanting to rely solely on the advances of computer hardware.

In heuristic search, abstractions of the problem space are defined as state-space abstraction functions (or mappings).

Definition 11 (State-Space Abstraction) *A* state-space abstraction ϕ *is a mapping from states in the original state-space* $\mathscr{S}$ *to the states in the abstract state-space* $\mathscr{A}$.

Abstraction mappings are, in a generic view, a common heuristic, as they are by definition a simplification of the problem's state-space, as seen in Figure 2.6. However, this area comes with its own trappings, as not all transformations are useful when searching for plans. For them to be reliably used in planning, these functions need to be *homomorphic*.

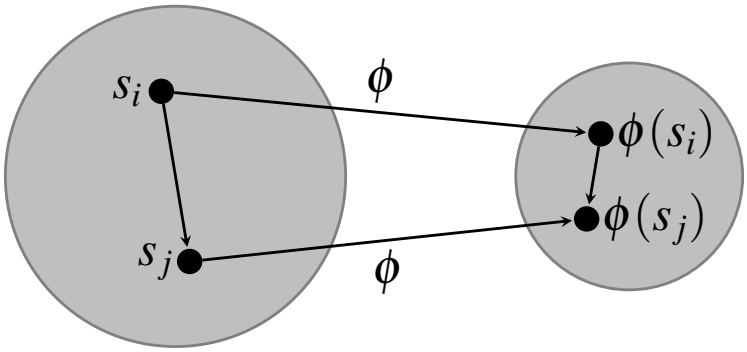


Fig. 2.6 Two states taken from the concrete state, with the abstraction transformation ϕ applied to get to an abstract space. In this figure, the edges are maintained, showing state-space homomorphism.

Definition 12 (State-Space Homomorphism) *A homomorphic abstraction* ϕ *imposes that if* s' *is the successor of* s *in the concrete state space we have* $\phi(s')$ *is the successor of* $\phi(s)$ *the in abstract one. This suggests that abstract operators* $\phi(o)$ *lead from* $\phi(s)$ *to* $\phi(s')$ *for each* $o \in \mathscr{O}$ *from* s *of* s'.

Homomorphic abstractions preserve the property that every path (i.e. plan) present in the original state-space is also present in the abstract state-space, and always shorter or the same length as prior to applying the abstraction. Still, abstract operators may yield spurious states, which we will touch on in the next section.

The abstractions vary depending on the way that state planning formalisms are in use. For example, if STRIPS is used, methods that remove a predicate (i.e. remove the predicate from the initial and goal states, and from all the operators pre- and post-conditions) from a state-space abstraction will be homomorphic. Methods such as PSVNs [72] use transformations such as *domain abstractions*, which consists of a mapping of labels $\phi : L \to L'$. This induces a state-space abstraction by modifying the labels of all constants in both concrete states and actions.

Definition 13 (Abstract Planning Task) *Let an abstract operator $o' = \phi(o)$ be defined as $pre' = \phi(pre)$, and $post' = \phi(post)$. For planning task $\mathscr{P} = \langle \mathscr{V}, \mathscr{O}, s_0, s_* \rangle$ with $s_0 \in \mathscr{S}$, $s_* \in \mathscr{S}^+$, the corresponding abstract task is $\langle \mathscr{V}, \mathscr{O}', s'_0, s'_* \rangle$ with $s'_0 \in \mathscr{A}$, $s'_* \in \mathscr{A}^+$, The result of applying operator $o' = (pre', post')$ to an abstract state $a = s'$ satisfying pre', sets $s'_i = post'_i \neq \sqcup$, for all $i = 1, \ldots, n$.*

As the planning problem spans a graph by applying a selection from a set of rules, the planning task abstraction is generated by abstracting the initial state, the partial goal state *and* the operators. Plans in the original space have counterparts in the abstract space, but not vice versa. Usually, the planning task of finding a plan from $\phi(s_0)$ to $\phi(s_*)$ in $\mathscr{A}$ is computationally easier than finding one from s_0 to s_* in $\mathscr{P}$.

2.5.4 Limitations

The main issue encountered when working with abstractions-based heuristics are *spurious paths*. They are paths that are formed in the abstract state-space that have no corresponding path in the original (concrete) state-space. An intuitive example of two disconnected paths $s_0 \to s_1 \to s_2 \to s_3 \to \ldots \to s_l$ and $s_{l+1} \to s_{l+2} \to s_{l+3} \to \ldots \to s_m = s_*$, is shown in Figure 2.7 with $l = 3$. As we map s_l and s_{l+1} to the same abstract state, we have an abstract plan which has no preimage in the original one.

The problem of spurious paths also appears in search spaces that have not been abstracted. One major cause for this is the process in which PDBs are created, by doing a regression search.

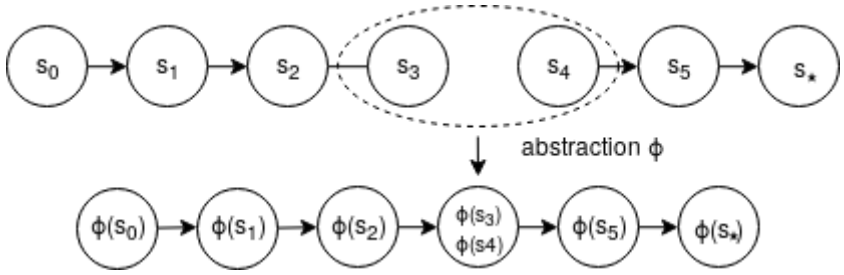


Fig. 2.7 Example of a spurious path problem.

To illustrate this issue, consider the (1×3) sliding-tile puzzle with two tiles 1 and 2 and one empty position, the blank, as an example (see Figure 2.8). In one SAS$^+$ representation, we have three state variables: two for the position of the tile $t_i \in \{1,2,3\}$, $i \in \{1,2\}$, and one for the position of the blank $b \in \{1,2,3\}$. Let $s_0 = (t_1,t_2,b) = (2,3,1)$ and $s_* = (1,2,\sqcup)$. The operators have preconditions $t_i = x$, $b = x+1$, and effects $t_i = x+1$, $b = x$, or preconditions $t_i = x$, $b = x-1$, and effects $t_i = x-1$, $b = x$, for $i = \{1,2\}$ and $x \in \{1,2,3\}$ (whenever possible). Going backwards from s_* the planner does not know the location of the blank and beside the reachable state $t_1 = 2$, $t_2 = 3$, $b = 3$ it generates two additional states $t_1 = t_2 = 1$, $b = 2$ and $t_1 = t_2 = 2$, $b = 1$.

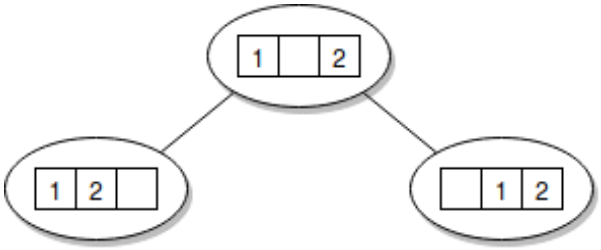


Fig. 2.8 Simple sliding-tile puzzle.

Research into mitigating this issue has led to the realization that we cannot guarantee the removal of all spurious states from an abstract state space, but it is possible to reduce their number. As we mentioned previously, in the sliding-tile puzzle there is a dual SAS$^+$ encoding with three variables denoting which tile (or blank) is present at a given position p_1, p_2, or p_3. This *exactly-one-of* state invariance is inferred by the static analyzer, but not used in the state encoding. This information can help reduce the number of spurious states.

Either spurious paths through abstraction or through regression, they do not affect the lower bound property of the resulting abstraction heuristic, but they can blow up the PDBs considerably, given that there are abstract states and paths for which no corresponding preimage in the forward space exists. As a result, refined state invariants (including mutually exclusive

detection of contradicting facts) greatly improve backward search and, thus, reduce the size of pattern databases, both in explicit and symbolic search, and is implemented in all of our planners.

2.6 Pattern Databases

In the field of AI Planning, Pattern Databases (PDBs) were introduced by Edelkamp [29], defining a pattern as a relaxation of the state, by selecting variables from the state-space while ignoring the others. However, the first to record impressive results using this approach were Culberson and Schaeffer [20, 21] in sliding-tile puzzles, in which a pattern consisted of a selection of tiles. Similar results were quickly replicated in a number of different domains that were based on combinatorial search. This led to providing the first optimal solutions of random instantiations of the Rubik's cube, with non-pattern labels being removed [84].

Definition 14 (Pattern Databases) *For a fixed goal state t and any abstraction space $S' = \phi(S)$, a pattern database is a lookup table indexed by $u' \in S'$, containing the shortest path from u' to the goal in abstract space $\phi(t)$. The size of a PDB is the number of states from S'.*

Definition 15 (PDB Creation) *A Pattern Database is created by conducting an exhaustive backwards search across the abstract state-space, usually via Breadth First Search (BFS) starting at $\phi(t)$. This assumes that for each action a, an inverse action a^{-1} can be derived, such that $v = a(u)$ if and only if $u = a^{-1}(v)$. In the case that the inverse actions $A^{-1} = \{a^{-1} | a \in A\}$ is equal to A, then the problem is reversible, leading to an undirected problem graph.*

2.6.1 Heuristic Details

As the planning task spans a graph by applying a selection of set of rules, the planning task abstraction is generated by abstracting the initial state, the goal state *and* the operators (modifying them according to the pattern in use). Important to note is that, while plans in the original space have counterparts in the abstract space, the opposite doesn't always hold true. This is important, as the main idea of the abstraction heuristics are that it is easier to find a plan by applying uniform blind search from $\phi(s_0)$ to $\phi(s_*)$ in the abstract state-space $\mathscr{A}$, compared to searching from s_0 to s_* in the concrete state-space $\mathscr{P}$.

Looking at Figure 2.9, we can see the same abstraction on two problems defined in the office domain. The first one, Figure 2.9a represents the abstraction on the problem defined initially, where Freddie, the robot assistant can move between all locations and has the transition diagram present in Figure 2.4. The second one, Figure 2.9b represents a slightly modified problem, where there is no path for Freddie between *wp2* and *wp3*. Both planning tasks will have the same abstract state-space.

PDBs are an admissible, consistent and additive heuristic. We can prove the first property as the lookup table consists of the optimal solutions in a relaxed state space. The consistency property is trivially shown by knowing that the shortest path distances satisfy the triangular inequality [29, 54].

2.6.2 Memory-based Heuristic

All efforts in creating the PDBs, by searching the abstract state-space, are spent prior to searching for a plan, so that these computations amortize through multiple lookups for the heuristic estimate of each state. However, this approach has also shown some major drawbacks: in the worst case, the time used in searching the relaxed state spaces may be more than the time saved searching the overall search space [138].

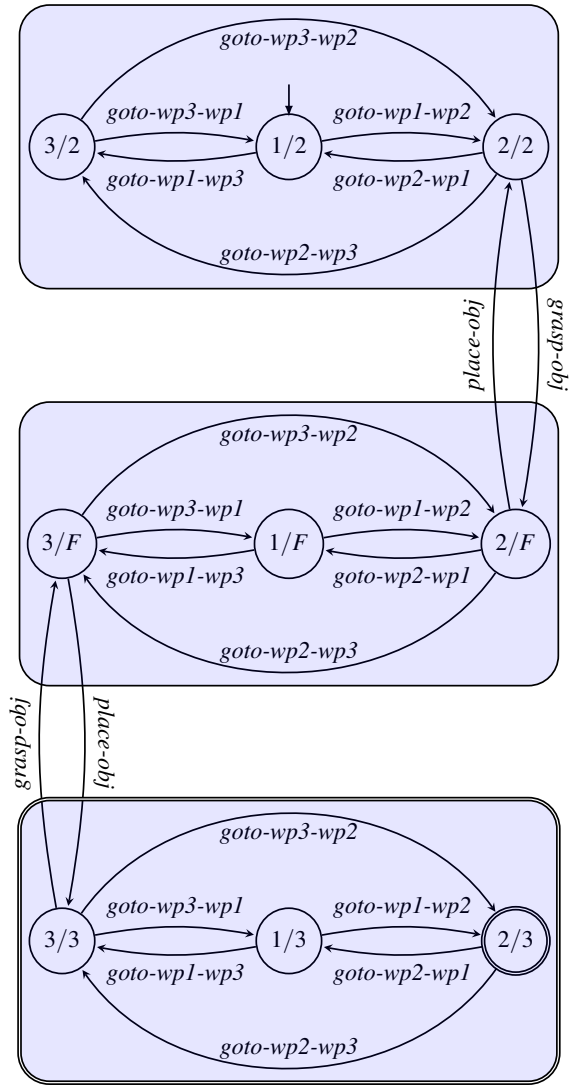
PDBs are not only a heuristic based on task abstraction, but are also memory-based, as all the work in constructing the heuristic (i.e. the lookup table) is done prior to concrete search. Therefore, it is vital to explore the relationship between the size of a pattern database and the number of nodes that will be expanded during search by an algorithm like $A^\star$.

Korf conjectures in [84] that when using this type of heuristic, $m \cdot t$ is constant, where m is memory and t is time. This conjuncture was confirmed by Holte et al. in [73], although in their results $log(t)$ and $log(m)$ are shown to be linearly related. This knowledge is important as it provides an assurance that, with an increase of memory available for the construction of the heuristic, search will be finished faster. In other terms, the more memory used when creating a PDB, the less time the A* algorithm will spend actually searching for a solution.

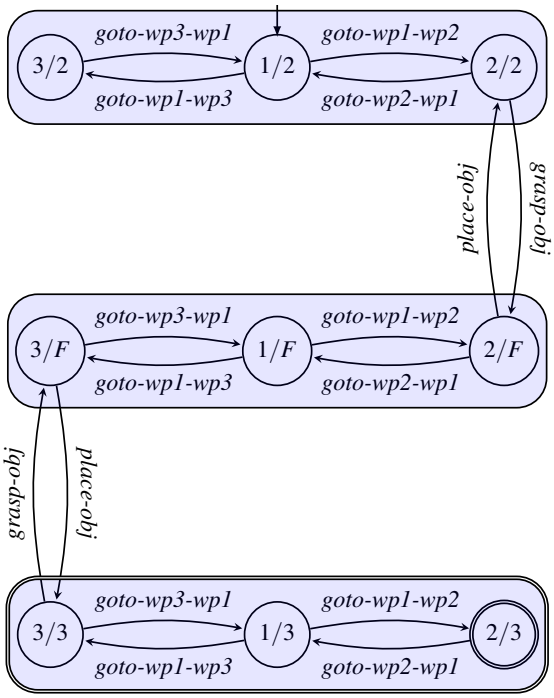
One way of applying PDBs to planning problems is by using the *Perimeter PDB*. In several planning tasks, generating the perimeter PDB finds already the plan, as shown by Franco et. al. [43]. As such, it can be used as a subroutine by a planner, allocating some time to generate a Perimeter PDB to see if it can be solved directly.

Definition 16 (Perimeter PDB) *A Perimeter pattern database is a PDB that results from the (blind) backwards shortest path search in the concrete state-space until memory resources are exhausted.*

For any state that has not been reached at once memory has ran out, the heuristic estimation is the maximum cost value found in the perimeter, while adding the cost of the operator with the smallest value .



(a) Projection on the *can_location* variable, applied on the initial example with all three locations interconnected.



(b) Projection on the *can_location* variable, applied on the initial example with all three locations interconnected.

Fig. 2.9 Abstract transition states for the office domain example, when projecting only on the variable *can_location*.

2.6.3 Using Multiple PDBs

It has also been shown that for PDBs the sum of heuristic values obtained via *projection* to a disjoint variable set is admissible [29]. The projection of state variables induces a projection of operators and requires *cost partitioning*, which distributes the cost $c(o)$ of operators o to the abstract state spaces [117]. We will discuss cost partitioning more in the following chapter.

Combining a group of PDBs into a single one has also been expressed as an optimisation problem, and for this we will need an objective function. For ease of notation, we identify a pattern database with its abstraction function ϕ.

Definition 17 (Average Fitness of PDB) *The average fitness f_a of a PDB ϕ (interpreted as a set of pairs $(a,h(a))$) is the average heuristic estimate $f_a(\phi) = \sum_{(a,h(a))\in\phi} h(a)/|\phi|$, where $|\phi|$ denotes the size of the PDB ϕ.*

There is also the option of evaluating the quality of a PDB based on a sample of paths in the original search space.

Definition 18 (Sample Fitness of PDB) *The fitness f_s of a PDB ϕ with regard to a given sample of (random) paths $\pi_1,\ldots,\pi_m$ and a given candidate pattern selection $\phi_1,\ldots,\phi_k$ in the search space is determined by whether the number of states with a higher heuristic value (compared to heuristic values in the existing collection) exceeds a certain threshold C, i.e.,*

$$\sum_{i=1}^{m}[h_\phi(last(\pi_i)) > \max_{j=1}^{k}\{h_{\phi_j}(last(\pi_i))\}] > C,$$

where $[cond] = 1$, if cond is true, otherwise $[cond] = 0$, and $last(\pi)$ denotes the last state on π.

Definition 19 (Pattern Selection) *The Pattern Selection Problem is to find a set of PDBs that fit into main memory, and maximize the average heuristic value.*

The average heuristic value has shown empirically that it is a good metric. While it is not the solution to evaluating the pattern selection problem perfectly, it is a good approximation up to this point.

2.6.4 What is a Pattern?

There exist several formal definitions of planning (e.g. STRIPS, PSVN and SAS$^+$), and the choice of which one is used will lead to a different concept of what a *pattern* is defined as. While at first glance a pattern would be defined as a selection of state variables, the concept of *pattern abstraction* is more general. In the case of combinatorial puzzles, a selection of state variables defines a pattern while in the case of relabelling, we generalize a pattern to the result of *domain abstraction* as described in Definition 20.

Definition 20 (Domain Abstraction) *Domain Abstraction ϕ is a mapping from D_i to some abstract set $A_i \subseteq D_i$, $i = 1, \ldots, n$, so that each concrete state s maps to some abstract state $a \in A_1 \times \ldots \times A_n$. As the preconditions and effects in SAS$^+$ planning are partial states, the mapping results in abstract operators.*

2.6.5 Implementing Patterns

There are two major techniques to generate admissible heuristics in the underlying state-space graph, namely node merging and edge insertion. For example, ignoring delete lists and the removal of preconditions leads to additional edges, while other abstraction techniques lead to merging of nodes, which can lead to multiple edges from one abstract state to another. While duplicate edges can be omitted in planning without cost, for planning tasks with cost, only the minimum cost of these edges preserves admissibility, hence guaranteeing optimal results.

On the other hand, omitting delete effects of actions, or alternatively the removal of preconditions, simplifies the planning task in order for it to be more practical to solve the problem in the abstract search space and use the resulting abstract optimal cost as an admissible heuristic. Mapping operators in SAS$^+$ for data abstraction is immediate, as all variable assignments in pre- and post-conditions can be assigned from the concrete value to the abstract one.

To prove that domain abstractions yield state-space homomorphism, we suppose that action $a = \phi(s)$ and state s' is the successor of s in $\mathscr{S}$. Then, the operator $\phi(o)$ is enabled in $\phi(s)$, as it fulfills $\phi(pre)$. Moreover, the result of applying the operator $\phi(o)$ to a yields $a' = \phi(s')$ as $post' = \phi(post)$.

To avoid the combinatorial explosion due to the consideration of all data abstractions available, the use of *don't care/undefined/*$\sqcup$ labels is used on our choice to ones that map all assignments of variables to *"don't care"* labels. This means that in practice, most of the variables will be not contribute to the computational effort for finding a solution to the abstract planning task. Variable projection is a special case of data abstraction and is used to generate pattern databases.

Definition 21 (Variable Projection) *Variable Projection is a data abstraction in which the abstract domain for every chosen variable is* $\{\sqcup\}$. *In this case a* pattern *is a selection of variables.*

2.6.6 Symbolic Pattern Databases

There has been considerable effort to show that PDB heuristics can be generated symbolically and used in a symbolic version of A* [27]. The concise representation of the Boolean formula for these characteristic functions in a binary decision diagram (BDD) is a technique to reduce the memory requirement during the search. Frequently, the running time for the exploration reduces as well.

2.7 Planning Applied to Robotics

As mentioned above, work in the late 1950s and 1960s at the Stanford Research Institute (SRI) resulted in STRIPS, the planning language, and the A* search technique [53] (described in algorithm 1) and leveraged the power of heuristics in guiding it to find a desired state. Both of these were developed to aid the first general-purpose robot, *Shakey the Robot*, which would be autonomous and be able to reason about their own actions.

This is an early example of the symbiotic relationship between the fields of AI Planning and Robotics, which has led to advances in both of the fields, with advances in one leading to benefits in the other.

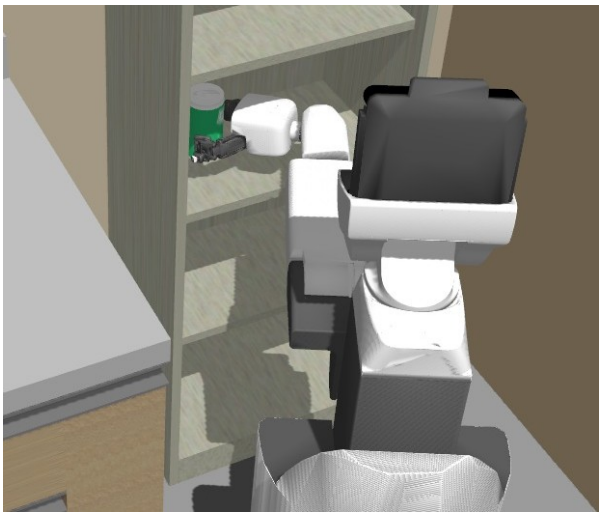


Fig. 2.10 The Robot Assistant Freddie from the Office domain, picking up a can while executing a plan.

2.7.1 SPA Paradigm

For a long time, robots have been designed using the three layered architecture [47, 1, 88], known as the Sense-Plan-Act (SPA) paradigm, with each one of the layers encapsulating a broad task that a robot needs to accomplish in order for it to successfully affect the environment it is in and accomplish complex tasks.

The first part of this paradigm is *sensing*, the component which is tasked with processing all the observations its sensors can gather from the environment in real-time. It also needs to monitor if the observations it is receiving are consistent with what it is expecting to happen when it applies an action in the environment that it is in. This is important as the environments most robots act in are dynamic and difficult to model, with a lot of uncertainty in them. Sensors are not perfect and are highly receptive to noise, which need to be addressed so that the reasoning process is not affected, otherwise it will lead to the agent not reaching its desired objectives.

Returning to the example of *Shakey the robot*, it was able to push a set of boxes around several interconnected rooms, following a plan Strings-generated by Planex, an execution monitor. During execution, members of the team would move the location of the boxes, changing what it was expecting the environment to look like. Planex would then be able to use information kept by the planner to recover from this failure [67].

The *planning* component will then receive all the observations from the sensors, and is tasked with finding a sequence of actions or a policy (i.e. a mapping of actions to states that will lead the agent to achieving its goals), while optimising a cost function defined by the user. The cost function depends on the planning task, as for some problems a user would want a robot to reach its goals in the shortest amount of time, while for others it could be distance traveled or battery used that are the most relevant factors.

Finally, the third component is called the *acting* or executing part. It is tasked with putting the plan into motion, executing each of the actions using the robot's actuators. It is necessary for each of the actions defined by the planning task to be defined and modeled based on a primitive action (usually defined as a state machine) inside the robot's control loop.

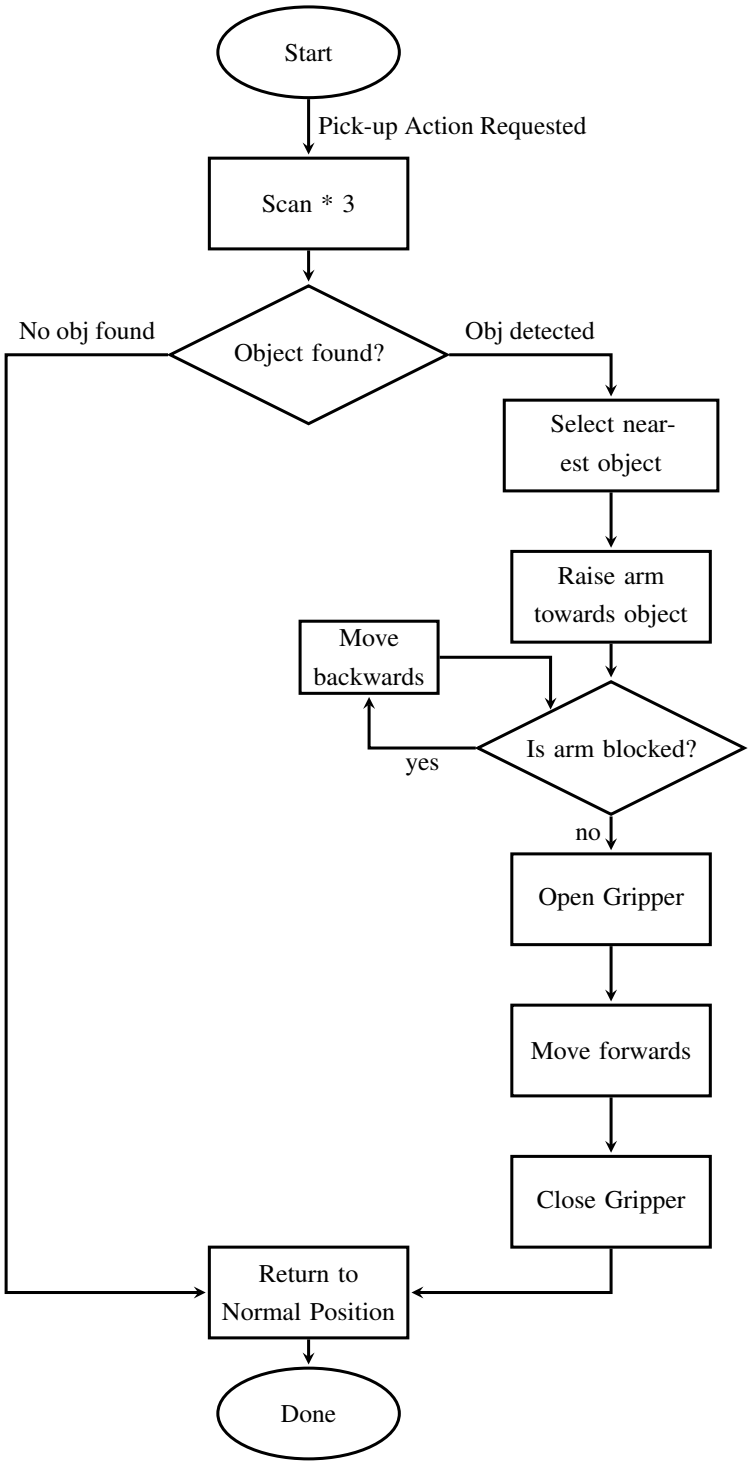


Fig. 2.11 State diagram for the `pickup` action.

An example of how an action could be defined in the robot, is the *pick-up* action from the office example. When the action is reached in the plan, the robot will follow the sequence in Figure 2.11.

Most robots used in research environments have run using the Robot Operating System, which we will briefly describe in the following section.

2.7.2 Robot Operating System

The Robot Operating System (ROS) consists of a set of open-source software tools and libraries that provide a well-defined communication layer above the host's operating system [120]. Software created for ROS is organized in `packages`, which consists of `nodes`, that communicate with each other via `messages`, that are sent with `topics`, and `services`:

- **Packages -** are the base of any software that can be developed in ROS, being the atomic unit of build and release. They might consist of nodes, other ROS-independent libraries, databases or other software that will be useful in the execution of the package. ROS was designed as such to ensure that each package could be easily used by other developers or users for their use-cases;
- **Nodes -** constitute a computational process, that are intended to run at a granular size. Best practice recommends for each node to fulfil only one role, similar to classes in Object-Oriented Programming (OOP). This approach has been used as it adds fault tolerance to the overall system, as crashes are contained to the scope of each node. One difference from classes in OOP languages is that nodes expose a limited API, as the intended way for them to communicate with other nodes is via `topics`, `services` and the `Parameter Server`;
- **Messages -** are a simple data structure that consist of typed primitives, such as integer, float, boolean etc., as well as arrays of primitive data structure types;
- **Topics -** are identifiable buses that are used for nodes to exchange messages, which were created for streaming information in only one direction. They use the anonymous publisher/subscriber paradigm, as this approach decouples the manner in which messages are created and how they are consumed by other nodes. The overall system works by having nodes that are interested in certain information subscribe to the relevant data,

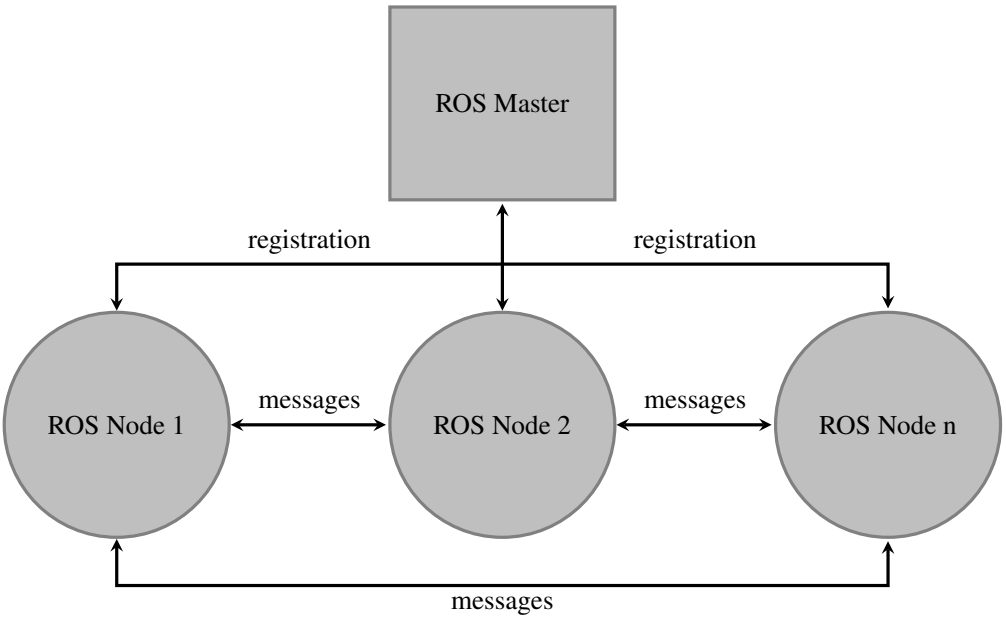


Fig. 2.12 A basic ROS system, which has *N* nodes registered to one ROS Master node (i.e. *roscore*). All the nodes communicate with each other via ROS messages. For this example, we can assume that all the nodes run on a single device (i.e. a robot), but they can run independently on different machines, as long as they are connected to the same network (or cloud-type infrastructure). This exposes the issue that if the machine running the ROS Master node stops, then the whole system will stop working.

while having the nodes that generate data publish it to the related topic. Topics have specific ROS message types;

- **Services -** similarly to topics, services are used to communicate information between nodes. However, they contrast from them by abandoning the publisher/subscriber paradigm, implementing instead the request/reply paradigm, leading to one node asking for some information and then waiting for a reply from a different node. They also use specific types that are defined in their ROS package.

Most ROS setups use either a single device (usually a robot or a laptop controlling the robot) or several devices that are all connected to the same master thread (called *roscore*). We can see an example of such a system in Figure 2.12, where there are *N* nodes connected to the same ROS master thread.

Over the years, there have been many new releases of ROS, which usually follows a major new releases of Ubuntu Long-Term Support (LTS) versions (the Unix-based Operating System they have been built on top). The last updated version of ROS was in 2020, when ROS Noetic

was released a month after Ubuntu 18.04. Since then, all focus has been on the newer version, called ROS 2 (first released in December 2018), with all new releases being brought for that version.

2.7.3 The ROSPlan Framework

Based on the ROS framework, Cashmore et. al. created ROSPlan[13], a standardized approach for integrating task planners into the Robot Operating System (ROS) and is shown in Figure 2.13. It consists of five nodes and several topics for each node:

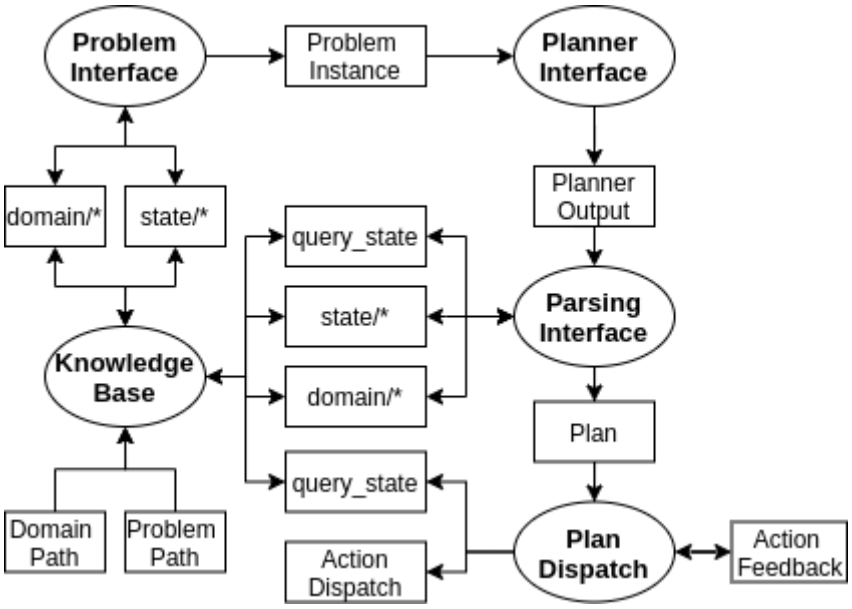


Fig. 2.13 Overview of the ROSPlan framework.

Knowledge Base

This component, seen in Figure 2.14, stores the PDDL model. It stores both a domain model and the current problem instance. It can do the following:

- Load a domain file (and optionally a problem) from file;
- Store the state as a PDDL instance;
- Update via ROS messages;
- Be queried for information.

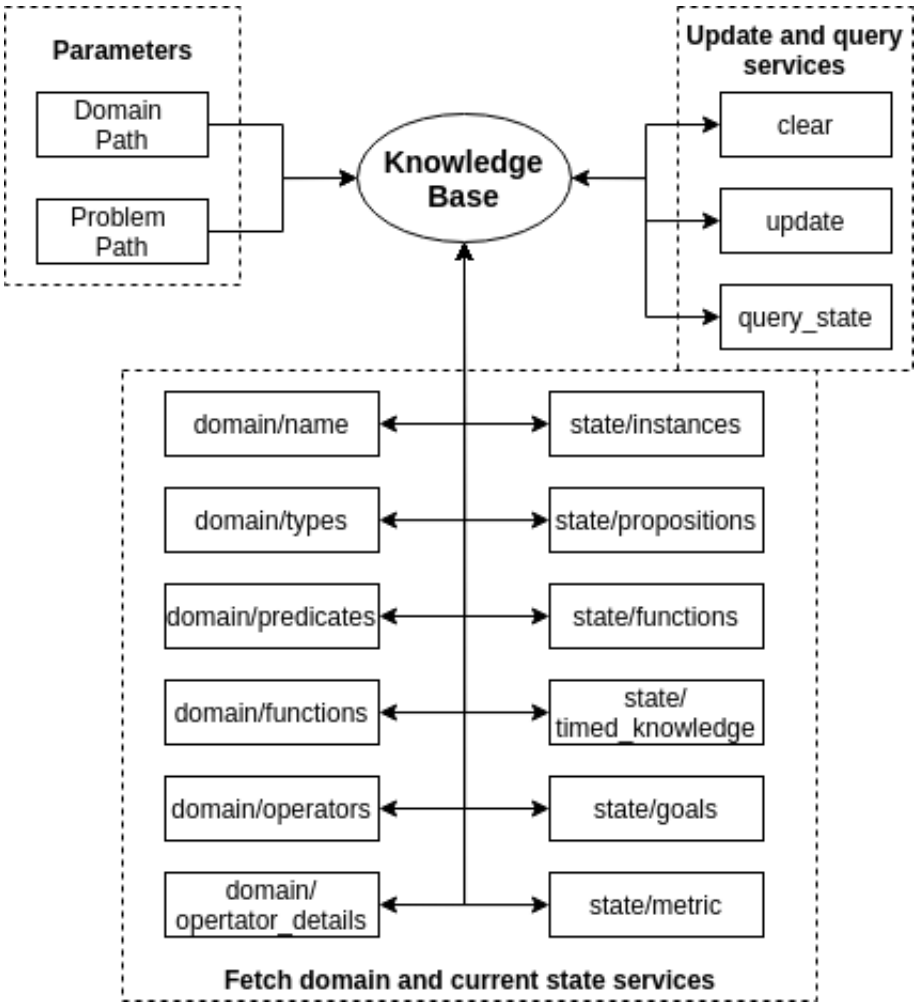


Fig. 2.14 Diagram representing the Knowledge Base of ROSPlan

Problem Interface

This node, seen in Figure 2.15, is used to generate a problem instance. It fetches the domain details and current state through service calls to a *Knowledge Base* node, and publishes a PDDL problem instance as a string, or write it to a file.

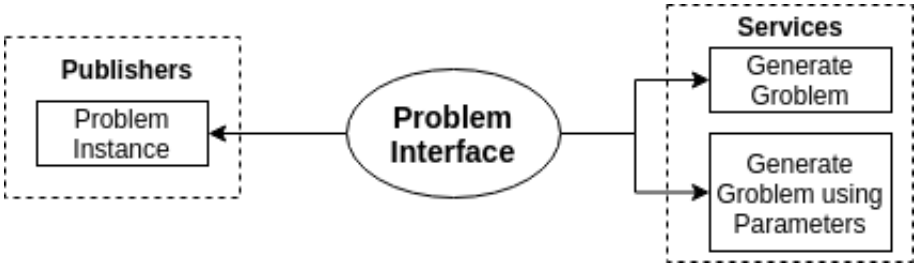


Fig. 2.15 Diagram representation of the Problem Interface

Problem Interface

The Planner Interface, seen in Figure 2.16, is a wrapper for an AI Planner. The planner is called through a service, which returns true if a solution was found to the problem sent. This interface feeds the planner with a domain file and a problem instance, and calls the planner with a command line specified via parameter.

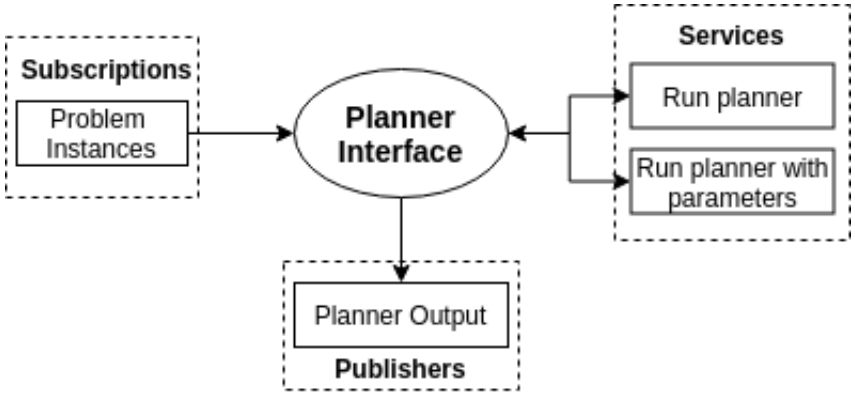


Fig. 2.16 Diagram representation of the Planner Interface

Parsing Interface

This node, seen in Figure 2.17, is used to convert planner outputs into a plane representation that can be executed, and whose actions can be dispatched to other parts of the system.

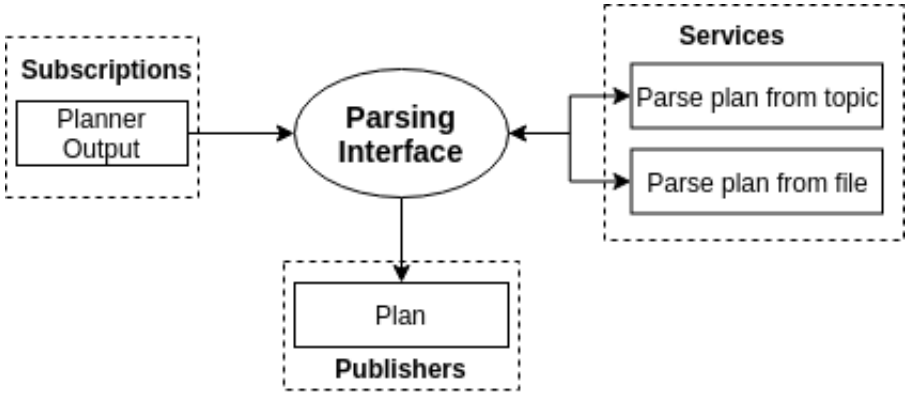


Fig. 2.17 Diagram representing the Parsing Interface

Plan Dispatch

This node, seen in Figure 2.18, includes plan execution, and is tasked with connecting atomic actions to the corresponding processes on the robotic systems that will fulfil them. An implementation of the Plan Dispatch node subscribes to a plan topic, and is closely tied to the plan representation of plans published on that topic.

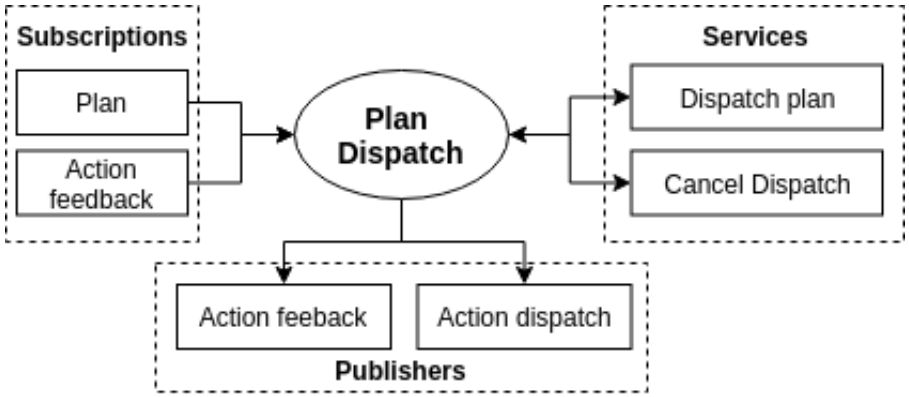


Fig. 2.18 Diagram representing the Plan Dispatch node

2.8 Belief-Desire-Intention Paradigm

The Belief-Desire-Intention model (BDI) model of agency is an architecture for practical reasoning. It was broadly motivated by Bratman's *Theory of Plans, Intentions and practical reasoning* [8].

A generic BDI Agent manipulates four different data structures:

- *beliefs*, comprising the information known by the agent, which is regularly updated as a result of agent perception;
- *plan library* or I-Plans, representing the behaviours available to the agent, combined with the situation in which they are applicable;
- *goals*, representing desired world states that the agent will pursue by adopting plans;
- *intention*, comprising a set of partially instantiated plans currently adopted by the agent.

Definition 22 (BDI Agent) *Is defined by the tuple* $\langle \mathscr{S}, \mathscr{A}, \mathscr{T}, \mathscr{B}, \mathscr{D}, \mathscr{I}, Del, \mathscr{M} \rangle$*, where:*

- $\mathscr{S}$ *is the state-space;*
- $\mathscr{A}$ *represents the set of actions;*
- $\mathscr{T}$ *is a function that, depending on the current state and action currently performed, will return the probability of reaching any of the other possible states;*
- $\mathscr{B}$, $\mathscr{D}$, $\mathscr{I}$ *represent the current beliefs, desires and intentions;*
- *a deliberation system Del*
- $\mathscr{M}$ *as a component for reasoning about actions.*

When comparing BDI agents to classical planning formalisms, state-spaces and actions are the same for both, and the transition function would be 1 or 0, as the agents are deterministic. The beliefs would be translated to the initial state of the planning problem, with the desires and intentions being the goal states, as in the case of planning problems, all the goals (i.e. desires) need to be fulfilled for the task to be completed. The deliberation and means-ends reasoning components would be the planner itself, which would need to find a solution to the task.

The behaviour of a BDI agent is prescribed in the set of plan rules, which is called a Plan Library.

Definition 23 (Plan Library) *A finite set of plan rules* $\{P_1, P_2, \ldots, P_n\}$ *is called a Plan Library. Each plan rule* P_i *is a tuple* $\langle t, c, bd \rangle$*, where:* t *is the invocation condition, an event indicating the event that causes the plan rule to be considered for adoption;* c *is the context condition, a first-order formula over the agent's belief base (with an implicit existential quantification); and* bd *is the plan body, consisting of a finite and possibly empty sequence of steps, where each step is either the invocation condition of a plan rule, or an action.*

2.9 Related works

In this section, we will go over the works related and relevant to our work and contributions presented across this thesis.

2.9.1 Pattern Database

Culberson and Schaeffer [21] extended the idea behind of abstraction heuristics to develop Pattern Databases, which proved to be the solution needed to find optimal solutions for the *fifteen-puzzle* [20]. In their solution, patterns were indicating which of the tiles could be abstracted away. This approach was then used by Korf to find optimal solutions for *Rubik's Cube* [84].

PDBs were first used in the context of classical AI planning problems by Edelkamp [29], which was done by projecting the state-space onto a memory limited abstract state space. This work was followed by extending the heuristic from explicit search to symbolic search [26], alleviating the memory demands of PDBs by leveraging a BDD description of the state-space.

Holte et al. [71] worked on methods to best use the memory available for creating PDBs, showing that creating several smaller PDBs and using the maximum value from all of them during search would get better results than generating only one. Research into this area has still not resulted into a definite answer, with methods on generating only one best PDB are still optimal on some domains, as showed by Kissmann and Edelkam [82] with the Gamer approach.

Also working with several PDBs, Korf and Felner [87] showed that, if the different goals can be divided into disjointed sets where actions in one would not affect another, then they could be all summed up and maintain the admissibility of the heuristic. This led to a greatly

improved performance on the fifteen-puzzle compared to using only one PDB. This work was then extended by Felner et al. [37] to other domains.

Research has since focused on finding methods to best generate and combine several PDBs while still maintaining the admissibility of the heuristic. The most influential methods for this have been the iPDB of Haslum et al., [55] and the GA-PDB by Edelkamp [28]. The first performs a hill-climbing search in the space of possible pattern collections, while the other employs a bin-packing algorithm to create initial collections, that will be used as an initial population for a genetic algorithm. iPDB evaluates the patterns by selecting the one with the higher h-value in a selected sample set of states, while the GA of the GA-PDB uses the average heuristic value as its fitness function.

Franco et al. [43] worked on combining iPDB and GA-PDB by generating one best collection of PBDs, called CPC, which showed a best single heuristic approach on the benchmarks when it was developed.

Pattern Databases have been extended to work on hybrid PDDL+ domains [114], by abstracting time and discretisation of the continuous state. Results showed that Temporal Pattern Databases (TPDB) and Partial TPDBs were competitive with the results of other PDDL+ planners [14, 113].

Another extension was in the area of probabilistic planning, where Klossner et al. [83] introduced MaxProb PDB, a PDB approach to solving the goal-probability maximisation (MaxProb) without relying on the determinisation of the probabilistic task. This was done by using the fact that abstract transitions of the abstract task have unique probabilities, resulting into an abstract planning task that is still a Markov Decision Process (MDP).

2.9.2 Planning with Uncertainty

Working with robotic agents leads to planning solutions needing to address the issues that appear when having uncertainty in the planning or execution phase. Uncertainty can appear either due to imperfect reading of sensors resulting into the agent being uncertain of its state or uncertainty in the execution due to actions not failing.

Solutions to this have varied from using different types of planning types (i.e. probabilistic planning, contingent planning), to approaches based on machine learning techniques. Planning in robotic agents took two main approaches, one by ignoring the action uncertainty and trying

to deal with how to act when it will fail, and the other approach was by incorporating the uncertainty in the planning model.

One approach on reasoning about an agent's execution when there is a high chance the plan might fail, is by preparing the system for failure and reasoning about replanning routines. This approach was first described by Hayes in [59], who proposed reusing the data structures (subgoal trees and decision graphs) that were used for the initial plan to guide replanning.

More recent work on replanning, Yoon et al. created FF-Replan [146], which is a planner that takes as input a probabilistic model, and then determinises the problem into a classical planning problem. It will find a plan for the problem by using the FF-planner [69], and if a failure happens during execution, it will plan again but in the same determinisation of the problem.

The solution presented by Gerard et al. [10] was to extend ROSPlan to accept any probabilistic planners based on RDDL models [126]. This way, uncertainty could be taken into account from the beginning, and instead of giving a plan as a solution, it would return a policy (mapping of states to actions).

Machine learning techniques have also been used, as done by Krivic et al. [89], who tackled the issue of uncertainty in the state the agent is in. Their solution was to approximate the values of the unknown planning variables by using Machine Learning classifiers trained on previous solutions.

BDI approaches differ from planning with uncertainty, as the agents are not expected to create neither a plan, nor a policy, but to select the appropriate Intention Plan (I-Plan) depending on the desires of the agent and the state of the environment. Our work focused on the idea of using I-Plans and the creation of plan libraries, for example using implementations such as PRS [77, 121] or Jason [7]. While most work using the BDI model has been in the context of software agents, it has also been used as part of a robot controller, for example in [109].

As [97] describes, most of the body of work in extending the BDI model initially described focuses on incorporating task planning to be used when no solution can be offered by a model. In those cases, the system relies on a task planner to come up with a solution via first principle planning (i.e. combining actions to make a plan for solving the planning task). These solutions are not compatible to be integrated into the BDI Plan Library as a new I-Plan, as they are specific for the task requested, whilst all the others are represented at a high level (i.e. general sequence of actions that can be applied on any situation when their context is compatible).

Exceptions to this are developed by Meneguzzi et al. and Srivastava et al. [98, 134]. The approach to the first one works by integrating a planning module and reintroducing the plan in the Plan Library. However, they limit themselves to classical planning models, and cannot support the expressiveness introduced in PDDL2 or later.

The reason for this lack of expressiveness is due to the computational intensity needed by such models, or by the difficulty of reusing a plan. In [105], results show that reusing or combining previously computed plans from a plan library is computationally complex. In particular, they proved that matching plan instances is NP-hard. As a result, our plan library is used to check if we already have a solution to a problem, not for planning from second principles. This has been one of the main reasons why for BDI agents combining plans is feasable, but it is not if the aim is to combine first and second principle task planning when coming from a typical planner.

2.10 Conclusion

In this chapter we offered all the prerequisite information necessary for understanding the works and contributions presented in the remaining of this thesis. We concluded it by having a review of works related to the ones we investigate and propose in this thesis.

In the following chapter we will investigate the first objective – **O1** – of this thesis, by investigating methods of solving cost-optimal planning problems with Pattern Databases. We will study the problem of pattern selection, for generating better heuristics from many smaller ones or by combining them with one larger PDB.

Chapter 3

Bin-Packing and Greedy Selection for PDB Creation

In this chapter, we will investigate our first objective – **O1** – by researching methods of solving cost-optimal planning with Pattern Databases. We will first explore ways to automatically generate patterns for planning problems. Following this, we will focus on how to achieve one best PDB, while still maintaining admissibility.

We will continue by introducing the pattern selection problem for working with several PDBs. Several methods have been proposed up to now on how to solve this selection problem such that one best admissible heuristic is created. We extend the methods proposed by Franco et al. [43], by modifying its pattern selection process, focusing on generating PDBs via Bin Packing and Greedy Selection.

Finishing the chapter, we will present the results for each of our proposed methods, showing how the results compare with other PDB based solutions. Work in this chapter has been published in [102, 104, 103].

3.1 Automated Generation of PDBs

Artificial Intelligence has long had the goal of developing methods for creating heuristics to help guide search algorithms without any help from the developer. This avenue of research has been going on from the early work of Gaschnik [46], Pearl [111], and Prieditis [118]. For

the field of AI Planning, Pattern Databases have been the closest approach to developing a domain-independent heuristic with a method of generating itself without any aid. PDBs have shown really good performance on an array of different domains, leading for it to become widely used in finding optimal solutions for different planning tasks.

3.1.1 Combinatorial Problems

As we discussed in the previous chapter, the initial results of Culberson and Schaeffer [21] in sliding-tile puzzles, where the concept of a pattern is a selection of tiles, quickly carried over to a number of other combinatorial search domains, leading to solving optimally random instances of the Rubik's cube, with non-pattern labels being removed [84].

For solving most combinatorial problems, such as the 15-puzzle, all the domain projections (i.e. the patterns used for the domain abstraction) were constructed manually by the developers. However, once the approach was seen to perform well in combinatorial optimisation, it was then implemented in the field of AI Planning by Edelkamp [29], and extended to work in a domain-independent approach, without needing to extend the modelling capabilities of PDDL. This then led to research focusing on automated creation of PDBs, and how a heuristic can be best created to solve as many planning tasks across different domains.

Introducing PDBs to solve planning tasks has led to another research question: should you create one or many PDBs to solve one task. In the rest of this chapter, we will go over our work in investigating how generating many smaller PDBs can be combined, by using different solutions to the bin-packing problem, and show that combining it with one greedily-constructed best pattern will result into state-of-the-art results.

3.2 Combining Multiple PDBs

The combination of several databases into one, is a difficult task, but it has usually led to better results compared to using only one. To maintain the admissibility of a PDB heuristic created from other smaller PDBs, initial work focused on combining several patterns by taking the maximum value out of each [71, 4].

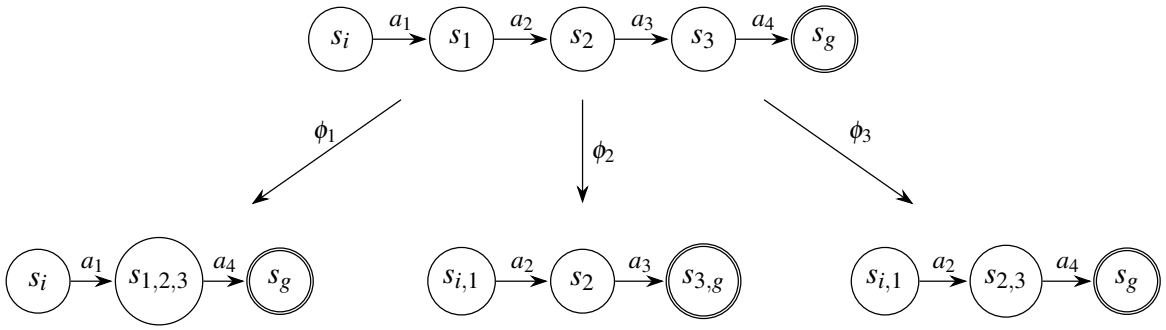


Fig. 3.1 We have a concrete state-space, with five states (s_i and s_g - the initial state and goal state respectively, s_1, s_2, s_3 three other states in the state) and four operators (a_1, a_2, a_3, a_4). Applying three different abstraction transformations (ϕ_1, ϕ_2 and ϕ_3), we get three different abstract state-spaces. All three could not be combined into one admissible heuristic, as it would lead to operators a_2 and a_4 being counted twice, and over anticipating the cost from s_i to s_g.

3.2.1 Disjointed Patterns

While the maximum of two PDBs always yields an admissible heuristic, the sum usually does not. This is due to having variables that are present in more than just one of the patterns and would lead to having the same operators (i.e. actions) counted multiple times when combining and summing up corresponding entries from each PDB. An example of this can be seen in Figure 3.1, where three abstractions have been applied on the concrete state-space, resulting into three different abstract state spaces. For us to maintain the admissibility of a heuristic based on all three abstractions ϕ_1, ϕ_2 and ϕ_3, it is necessary to use the max value, otherwise the cost of a_2 and a_4 will be counted twice.

Korf and Felner [87] showed that with a certain selection of disjoint patterns, the values in different PDBs can be added while preserving admissibility. This is done not only by selecting patterns that do not have any overlapping variables, but also by having abstract operators that have no effects on variables present in the other pattern. In Figure 3.1, two disjoint (additive) patterns would be ϕ_1 and ϕ_2.

Definition 24 (Disjoint State-Space Abstractions) *Two state-space abstractions ϕ_1 and ϕ_2 are* disjointed*, iff for all abstract actions a' and a'' generated by ϕ_1 and ϕ_2 respectively, the relationship*

$$\phi_1^{-1}(a') \cap \phi_2^{-1}(a'') = \emptyset, where \quad \phi_1^{-1}(a') = \{a \in A | \phi_i(a) = a'\} \quad for \quad i = 1,2$$

is true.

Further research into this led to Holte et al. [71] indicating that several smaller PDBs could perform better when compared to one large PDB, as it was quicker and more memory efficient to build many small ones. Haslum et al. [55] built on this idea by introducing the *canonical heuristic* and pattern collections.

3.2.2 Pattern Collections

Haslum et al. defined it as a collection of patterns $\mathscr{C}$, which could be additive if and only if all the patterns of $\mathscr{C}$ are pairwise disjoint. This is most of the time not the case, leading to the sum over $\mathscr{C}$ as not being an admissible heuristic. However, there could be subsets of $\mathscr{C}$ could be additive. The canonical heuristic is then defined as the maximal additive subset (MAS) of $\mathscr{C}$:

$$h^{canon}_{\mathscr{C}}(s) = \max_{A \in MAS(\mathscr{C})} \sum_{P \in A} h^P(s)$$

Calculating $MAS(\mathscr{C})$ is an NP-complete problem, as it entails looking for the maximal clique between each pair of additive heuristics in a collection. However, if the number of patterns is small, it is a practical solution to generating good PDBs from many smaller patterns.

This line of reasoning has changed the focus to creating methods that would select good pattern collections, with proposed methods ranging from evaluating a pattern based on its average heuristic value [28], to sampling the search space of possible new variables that could be added to the pattern [55].

Many planning problems can be translated into state-spaces of finite domain variables [60], where a selection of variables (pattern) influences both states and operators. For disjoint patterns, an operator must distribute its original cost, if present in several abstractions [80, 145].

In a domain-independent planning scenario, it is extremely difficult to maintain fully disjointed PDBs, as operators are more interconnected and would lead to either having a small number of abstractions, or losing the admissibility of the heuristic. This leads to the need for partitioning operator cost across several patterns.

Franco et al. [43] proposed a method that focuses on creating collections of PDBs that *complement* each other (CPC), showing the best results compared to the other methods described. We will focus mostly on this approach, as they showed the most space for performance

improvement possible, by selecting the best combination of patterns, while mostly ignoring the cost-partitioning problem.

3.3 Pattern Selection and Cost Partitioning

The automated selection of the most informative patterns remains a combinatorial challenge and a major ambition in the field of heuristic search. There is an exponential number of variable sets to choose from, not counting alternative projection and cost partitioning methods [79] in distributing the cost of actions over different abstract search spaces.

3.3.1 Cost Partitioning

Using multiple abstraction heuristics can lead to solving more complex problems, but to preserve optimality, we need to distribute the cost of an operator among the abstractions. One way of doing this, Saturated Cost Partitioning (SCP), is presented by Seipp and Helmert [128]. SCP has shown benefits and often better results to simpler cost partitioning methods, being proven that it dominates these simpler methods [130]. Given an ordered set of heuristics, in our case PDBs, SCP relies on only using those costs which each heuristic uses to create an abstract plan. The remaining costs are left free to be used by any subsequent heuristic. However, considering the limited time budget, this approach is time-consuming compared to other cost partitioning methods [116].

Greedy 0/1 cost partition, zeroes any cost for subsequent heuristics if the previous heuristic has any variables affected by that operator. Both SCP and 0/1 allow heuristics values to be added admissibly. SCP dominates 0/1 cost partitioning (given a set of patterns and enough time, SCP would produce better heuristic values), but it is much more computationally expensive than 0/1 cost partitioning.

Franco et al., [43] shows that, in order to find good complementary patterns, it is beneficial to try as many pattern collections as possible. As such, we implemented 0/1 cost partitioning in all of our work. We evaluated this using the canonical cost partitioning method [56] as well whenever we added a new PDB, but this resulted in a very pronounced slow down which increased the number of PDBs that were already selected. This was the reason we adopted a hybrid combination approach, where 0/1 cost partition is used on-the-fly to generate new pattern collections, and, only after all interesting pattern collections have been selected, we run

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
PDB_1	0	0	0	0	0	1	0	1
PDB_2	0	1	0	0	0	0	1	0
PDB_3	0	1	0	0	1	0	0	0
PDB_4	0	0	1	1	0	0	0	0

Table 3.1 An example set of pattern (database) variable selection, forming a 0/1 GA bitstring (or a solution of the bin packing problem).

the canonical combination method, slightly extending to take into account that each pattern has its own 0/1 cost partition.

3.3.2 Genetic Algorithms Pattern Selection

A *genetic algorithm* [70] is a very general optimisation method, and member of the class defined as *evolutionary strategies*. It refers to the recombination, selection, and mutation of "genes" (states in a state-space) to optimize the "fitness" (objective) function.

In a genetic algorithm (GA), a population of candidate solutions to an optimisation problem is sequentially evolved to generate better solutions. Each candidate solution has a set of properties (its "chromosomes") which can be mutated and altered. Traditionally, solutions are represented as 0/1 bitvectors.

An early approach for the automated selection of PDB variables by Edelkamp (2007) employed a GA with genes representing state-space variable patterns in the form of a 0/1 matrix G, where $G_{i,j}$ denotes that state variable i is chosen in PDB j (see Table 3.1). Besides changing bits, mutations may also add and delete PDBs in the set.

To evaluate the fitness function, the corresponding PDBs had to be generated – a time-consuming operation, which nevertheless pays off in most cases. The approach has been refined by Lelis et al. and 2016 Franco et al. [91, 43] and is now available in the fast-downward planning system [61].

The PDBs corresponding to the bitvectors in the GA have to fit into the main memory, so we need to restrict generating offspring. An alternative proposed by Edelkamp [28], which is also used as a subroutine for the GA, is to cast the pattern selection as a bin packing problem, which has for long been research and can offer many fast solutions to this task. For this reason,

research in PDBs has kept relying on this approach for developing better heuristics [43] and we will describe in the following subsection this approach in more detail.

3.3.3 Bin Packing for Pattern Selection

The bin packing problem (BPP) is one of the first problems shown to be NP-hard [45]. Given objects of integer size $a_1, \ldots, a_n$ and maximum bin size C, the problem is to find the minimum number of bins k so that the established mapping $f : \{1, \ldots, n\} \to \{1, \ldots, k\}$ of objects to bins maintains $\sum_{f(a)=i} a \leq C$ for all $i \leq k$. The problem is NP-hard in general, but there are known approximation strategies such as first-fit and best-fit decreasing (being at most 11/9 off the optimal solution [24]. The NP reduction is from number partitioning (where objects have to be split into two equally sized parts): if $\sum_{i=1}^{n} a_i$ is odd, then number partitioning is not solvable; if $\sum_{i=1}^{n} a_i$ is even, then the objects have a perfect fit into two bins of size $\sum_{i=1}^{n} a_i/2$.

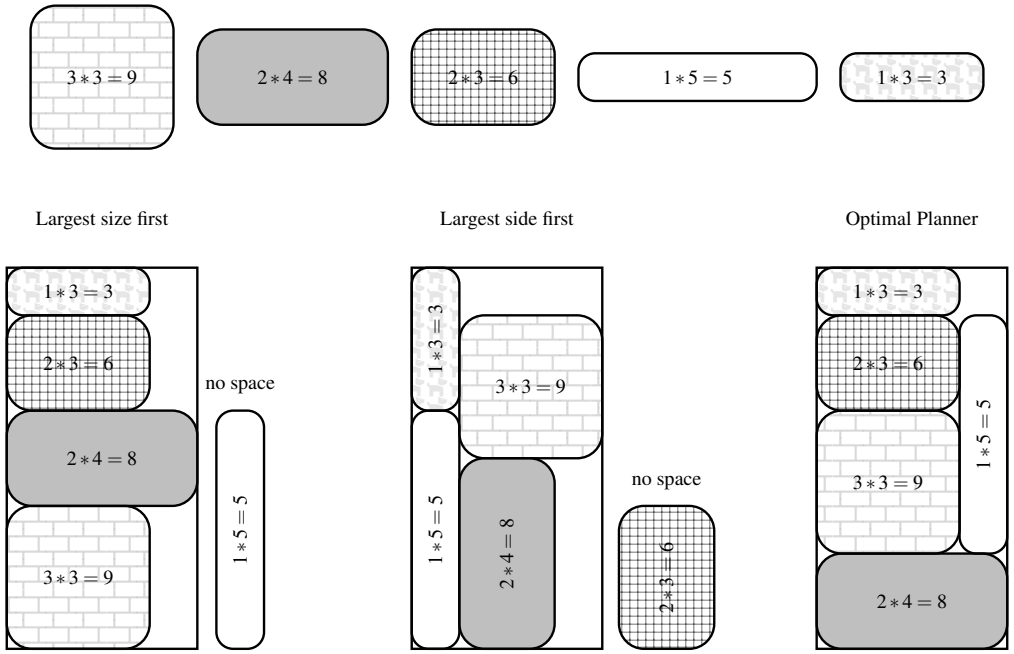


Fig. 3.2 Example of a two-dimensional bin packing problem, with three different methods of solving it: one by using the largest size first, one by ordering in function of the largest side, and the last by an optimal planner.

In the PDBs selection, the BPP is slightly different. In this situation, we will be using bins as PDBs (the term pattern is also used), and we think of variables or smaller previously computed PDBs as items we will try to fit in the bin. We estimate the expected size of the PDB by computing the product (not the sum) of the domain sizes, so that for a maximum

bin capacity M imposed by the available memory, we find the minimum number of bins k so that the established mapping f of objects to bins maintains $\prod_{f(a)=i} a \leq M$ for all $i \leq k$. By taking the logs of both sides, we are back to sums, but the sizes become fractional. In this case, $\prod_{f(a)=i}$ is an upper bound on the number of abstract states needed.

Taking the product of variable domains is a coarse upper bound. In some domains, the abstract state spaces are much smaller. Bin packing chooses the memory bound on each individual PDB, instead of limiting their sum. Moreover, for symbolic search, the correlation between the cross product of the domains and the memory needs is weak.

As bin packing is *pseudo-polynomial*, small integers in the input lead to a polynomial-time dynamic programming algorithm [45]. There is a *bin completion* strategy that is featured in depth-first branch-and-bound algorithms for bin packing described in [85] and [86]. The key property that makes the bin completion efficient is a dominance condition on the feasible completions of a bin. The algorithm that partitions the objects into included, excluded and remaining ones relies on perfectly fitting elements and forced assignments, and thus, on integer values for a. In the given setting of real-valued object sizes that are multiplied (or logarithms that are added), this might be less often the case.

By limiting the amount of optimization time for each BPP, we do not insist on optimal solutions, but we want fast approximation strategies that are close-to-optimal. Recall that suboptimal solutions to the BPP do not mean suboptimal solutions to the planning problem. In fact, *all* solutions to the BPP lead to admissible heuristics and therefore optimal plans.

3.4 Pattern Selection Improvements

For the sake of generality, we strive for solutions to the problem, which do not include problem-specific knowledge but still work efficiently. Using a general framework also enables us to participate in future solver developments. Therefore, we decided for the moment to focus on the First-Fit on-line algorithm[1].

[1] Even though in principle, BPP can be specified as a PDDL planning problem on its own, initial experiments of solving such a specification with off-the-shelf-planners were not promising.

3.4.1 First-Fit Increasing/Decreasing

First-Fit Increasing (FFI), or Decreasing (FFD), is a fast on-line approximation algorithm that first sorts the objects according to their sizes and, then, starts placing the objects into the bins, putting an object in the first bin it fits into. In terms of planning, the variables are sorted by the size of their domains in increasing/decreasing order. After that is done, the *first* variable is chosen and packed at the same bin with the rest of the variables which are related to it if there is enough space in the bin. This process is repeated until all variables are processed.

For the sake of completeness, we provide its implementation.

Listing 3.1 A First-Fit Increasing algorithm implemented into C++. It will initially order all the variables, and in order of size it will add them to a bin. If one variable doesn't fit in a bin, it will be placed in the next one that has space for it.

```
int firstfit() {
  int c=0; double bin[n];
  for (int i=0;i<n;i++) bin[i] = C;
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
      if (bin[j]-a[i] >=0) {
        bin[j]-=a[i]; break; }
  for(int i=0;i<n;i++)
    if (bin[i] != C) c++;
  return c;
}
```

It is straight-forward to extend this algorithm to the multiplication of object sizes and to long integer and floating point arithmetics.

3.4.2 Constraint Programming

Constraint programming [92] is a research field of its own, with a huge body of work and results of a huge quality in certain domains where there could be many possible solutions to one problem. Depending on the encoding of the underlying optimization problem, however, it shares many links to AI search and planning. In this former field, there are planners like SATPlan [81], ones based on integer programming [141], SMTPlan [12] or Constraint Satisfaction Problems

[139], that compile a planning problem into a SAT, IP, SMT or CSP problem, leaving the native solver to resolve the problem and then parse it back to a plan.

In our work on solving the BPP, we initially looked at finding solutions quickly by using suboptimal approximations. However, we wanted to look at how results would change if the Pattern Selection would be optimised based on memory, by trying to get as many variables possible in each pattern database. By doing this, our hypothesis was that it would increase the chances of having more important variables for solving the task in any given PDB, and as this could have a huge pool of solutions, Constrain Programming would cater well for this task.

For our work, we have integrated into the Fast Downward implementation of the CPC heuristic [43], MiniZinc [107], a solver-independent declarative modelling language for Constraint Programming. We defined Pattern Selection as a bin packing problem, which would be parsed into MiniZinc format and submitted to a CP solver. The solution would then be used as the final pattern for our heuristic.

Listing 3.2 The constraints definition that is used by the MiniZinc solver for declaring the Pattern Selection problem into a Bin-Packing Problem.

```
int: num_bins; int: num_objects;
array[1..num_objects] of int: object;
int: bin_capacity;
array[1..num_bins, 1..num_objects] of var 0..1: bins;
array[1..num_bins] of var 0..bin_capacity: bin_loads;
var 0..num_bins: num_loaded_bins;

constraint
  forall(b in 1..num_bins)
   (bin_loads[b] = sum(s in 1..num_objects)
   (object[s]*bins[b,s])) /\
  sum(s in 1..num_objects) (object[s]) =
   sum(b in 1..num_bins) (bin_loads[b]) /\
  forall(s in 1..num_objects)
  (sum(b in 1..num_bins)(bins[b,s]) = 1) /\
  decreasing(bin_loads)  /\ bin_loads[1] > 0 /\
  num_loaded_bins = sum(b in 1..num_bins)
  (if bins_of_load[b] > 0 then 1 else 0 endif);
```

```
solve minimize num_loaded_bins;
```

CP for Pattern Selection

Recent work on constraint programming for bin packing includes [22]. Our intent for solving the BPP is to be solver-independent, and to apply descriptive models. The aforementioned MiniZinc is compiled into the input of a range of different constraint-based solvers including Gurobi [52] and Gecode [48]. The BPP model is defined in Listing 3.2. In our implementation, we found using Gurobi would offer better results, but we will need a more thorough investigation with many more solvers for a conclusive result.

It requires that each $object[i]$ with a specific weight that is held in the variable is put into a bin. The 2-d set $bins[i, j]$ displays 1 only if object j is placed in bin i, otherwise it holds the value 0. The model itself is quite straightforward, using only 6 constraints, 2 of which are put in place to help break symmetry.

3.4.3 Greedy Selection

Franco et al. [43] compared the pattern selection method to the Gamer approach [82], which tries to construct one single best PDB for a problem. Its pattern selection method is an iterative process, starting with all the goal variables in one pattern, where the causally connected variables, who would most increase the average h value of the associated PDB, are added to the pattern.

Following this work, we devised a new *Gamer-style* pattern generation method, which behaves similarly, but which adds the option of *partial pattern database* generations to it. By partial we mean that we have a time and memory limit for building each PDB. If the PDB building goes past this limit, we truncate it in the same way we would do with a perimeter PDB, i.e. any unmapped real state has the biggest h value when the PDB creation was interrupted.

An important difference with the Gamer method is that we do not try every possible pattern resulting in an addition of a single causally connected variable to the latest pattern.

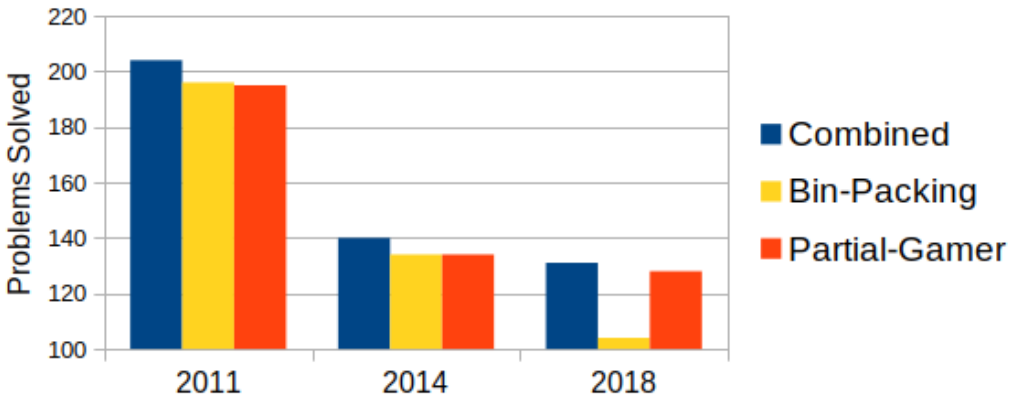


Fig. 3.3 Coverage of Bin Packing, Partial Gamer and of both combined on three latest cost-optimal IPC benchmark problems.

3.5 Symbolic PDB Planners

Based on the results from [43], we decided to work only with Symbolic PDBs. Further, our experiments suggested that the PDB heuristic performs best when it is complemented with other methods. One good combination was using our method to complement a symbolic perimeter PDB method that we used in the first planner we present. The selected method to be complemented first generates a symbolic PDB up to a fixed time limit and memory limit. One advantage of seeding our algorithm with such a perimeter search is that if there is an easy solution to be found in what is essentially a brute force backwards search, we are finished before even creating a PDB.

Secondly, we combined the Partial-Gamer with bin packing and saw very good coverage (i.e. number of problems solved across total number of experiments attempted) across in how they complemented each other. In Figure 3.3 we see that each method gives good results on their own, Bin-Packing solving 434 instances and Partial-Gamer 457, but when used together they increased to 475.

In our work, however, we decided to use a hybrid approach to the state-space description, keeping the forward exploration explicit-state, and the PDBs generated in the backward exploration symbolic. Lookups are slightly slower than in hash tables, but they are still in linear time to the bitvector length.

In this section, we will present three symbolic planners, Planning-PDBs, MiniZincPDB and GreedyPDB, based on the Fast-Downward [Helmert, 2006] planning framework, and extensions of the CPC heuristic [43]. The two differ in the pattern selection methods that we use in each of them, the first two having a similar structure but differing in their bin-packing solution, while the last works by combining the Greedy Selection method with bin-packing.

3.5.1 Planning-PDBs

In *Planning-PDBs*, we start with the construction of the perimeter PDB, and continue by using two bin-packing methods to create a collection of PDBs. The first method uses first-fit increasing, while the second being first-fit decreasing. Bin-packing for PDBs creates a small number of PDBs which use all available variables. Even though reducing the number of PDBs used to group all possible variables does not guarantee a better PDB, by having smaller PDB collections, it is less likely to miss interactions between variables due to them being placed on different PDBs. The bin packing algorithms used ensures that each PDB has at least one goal variable.

If no solution is found after the perimeter PDB has been finished, the method will start generating pattern collections stochastically until either the generation time limit or the overall PDB memory limit is reached. We then decide whether to add a pattern collection to the list of selected patterns if it is estimated that adding such PDB will speed up search. We optimize the results given by the bin-packing algorithm by giving it to a GA. It then resolves operator overlaps in a 0/1 cost partitioning. To evaluate the fitness function, the corresponding PDB is built —a time-consuming operation, which nevertheless paid off in most cases. Once all patterns have been selected, the resulting canonical PDB combination is used as an admissible heuristic to do A* search.

3.5.2 MiniZincPDB

We implemented a Pattern Selection approach that implemented a CP model for its bin packing subroutine, calling it MiniZincPDB [2]. It is the exact same planner as Planning-PDBs except for the way it solves the bin packing problem. The results of the CP solver are then entered as the

[2]We chose Gecode as the solver this time, we are planning on extending to more solvers

initial population for the GA that then will optimize for a duration of 900 seconds (time found empirically to give the best results).

3.5.3 GreedyPDB

We encountered that greedily constructed PDBs outperform the perimeter PDB, which we decided not to use. The two construction methods do not complement well, on the extreme case greedy PDBs will build a perimeter PDB after adding all the variables. There is a significant amount of overlapping between both methods. The collection of patterns received from bin packing, however, complements the greedily constructed PDBs well. One reason for this is that in domains amenable to cost-partitioning strategies, i.e. alternative goals are easily parallelized into a complementary collection of PDBs, bin packing can do significantly better than the single PDB approach. Evaluation is based on the definition of sample fitness. The sample is redrawn each time an improvement was found.

Algorithm 2 shows how Greedy PDBs combines two bin packing algorithms with a greedy selection method called Partial Gamer. The two bin packing algorithms use First Fit Decreasing (*FFD*) and First Fit Increasing (*FFI*), same used in BP-PDB. For FFD we set a limit of 50 seconds, while for FFI we used a limit of 75 seconds (both limits were found empirically to give the best results). To evaluate (*EM*), if the generated pattern collections should be added to our selection ($\mathscr{P}_{sel}$), we used a random walk as an evaluation method. If enough of the sampled state heuristic values are improved, the pattern is selected. Partial Gamer greedily grows the largest possible PDB by adding causally connected variables to the latest added pattern. If a pattern is found to improve, as defined by the evaluation method, then we add it to the list of selected pattern collections as a pattern collection with a single PDB. Note that we are using symbolic PDBs with time limits on PDB construction, hence a PDB which includes all variables of a smaller PDB does not necessarily dominate it since the smaller PDB might reach a greater depth.

An important difference with the Gamer method is that we do not try every possible pattern resulting in the addition of a single causally connected variable to the latest pattern. As soon as a variable is shown to improve the pattern, we add it and restart the search for an even larger improving pattern. We found this to work better with the tight time limits required by combining several approaches. All the resulting pattern database collections are combined by simply maximizing their individual heuristic values. The PDBs inside each collection were combined using zero-one cost partitioning. The rationale behind the algorithm is that some

domains are more amenable to using several patterns where costs are distributed between each pattern, while other domains seem to favor looking for the best possible single pattern.

Algorithm 2 Greedy PDBs Creation

Require: time and memory limits T and M, min and max PDB size S_{min} ad S_{max}, evaluation method EM.

function GREEDYPDBS(M,T,S_{min},S_{max},EM) :
    $SelPDBs \leftarrow \emptyset$
    $\mathscr{P}_{sel} \leftarrow \mathscr{P}_{sel} \cup Packer(FFD, S_{min}, M, T, EM)$
    $\mathscr{P}_{sel} \leftarrow \mathscr{P}_{sel} \cup Packer(FFI, S_{min}, M, T, EM)$
    $\mathscr{P}_{sel} \leftarrow \mathscr{P}_{sel} \cup PartialGamer(M, T, EM)$
    Return $\mathscr{P}_{sel}$
end function

Algorithm 3 Packer

function PACKER($Method$,S_{min}, M, T,EM) :
    $SizeLim \leftarrow S_{min}$
    while $(t < T)$ **and** $(m < M)$ **do**
        GENERATE_$\mathscr{P}$($Method$,$SizeLim$)
        if $EM(\mathscr{P})$ **then**
            $\mathscr{P}_{sel} \leftarrow \mathscr{P}$
        end if
        $Size \leftarrow Size * 10$
    end while
    Return $\mathscr{P}_{sel}$
end function

3.6 Experiments

Following is an ablation-type study where we analyzed which components worked best. We ran different configurations on all the planning benchmarks (from all the previous IPCs) on our cluster that has 26 Intel Xeon E5-2660 V4 (2.00GHz) processors and 192 GB of RAM. We compare GreedyPDB, Planning-PDB and MiniZincPDB with other pattern database planners – Complementary1 and 2 [43, 42], Scorpion [128] – and the symbolic benchmark planner – SYM-BiDir [32] – that took part in the cost-optimal track from the 2018 International Planning Competition. For the first part, we focused on the results from the most modern benchmark

Algorithm 4 Partial gamer

```
1: function PARTIALGAMER(M, T, EvalMethod) :
2:     InitialPDB ← all goal variables
3:     SelPDBs ← InitialPDB
4:     while (t < T) and (m < M) do
5:         generate all CandidatePatterns resulting of adding one casually connected variable
   to latest P ∈ 𝒫_sel
6:         for all P ∈ CandidatePatterns do
7:             if EM(𝒫) then
8:                 𝒫_sel ← P
9:                 break
10:            end if
11:        end for
12:    end while
13:    Return 𝒫_sel
14: end function
```

sets, and the normalised-coverage results were taken from all the domains available. We will discuss more on the reason of this separation in Section 4.5.

Year/Method	2011	2014	2018	Total
GreedyPDB	204	140	**131**	475
PlanningPDB	190	131	123	444
MiniZincPDB	**216**	**156**	123	**495**
Scorpion	190	118	104	412
SymBiDir	174	129	114	417
Comp1	185	111	123	419
Comp2	204	155	124	483
Oracle	227	171	143	541

Table 3.2 Overall coverage of PDB-type planners across different International Planning Competitions for cost-optimal planning.

Looking at the results of various cost-optimal planners across all domains from the IPC competitions from 2011 to 2018 in Table 3.2, we get a good overall picture on the PDB planner performance. Symbolic bidirectional search (417 problems being solved) is almost on par with Scorpion (412) and Complementary1 (419), while PlanningPDB performs slightly better (444). The overall top three planning systems are GreedyPDB (475), Complementary2 (483) and MiniZincPDB (495).

Domain/Method	Agr	Cal	DN	Nur	OSS	PNA	Set	Sna	Spi	Ter	Total
GreedyPDB	13	**12**	14	**15**	**13**	16	8	13	11	16	**131**
Planning-PDB	6	12	14	12	**13**	**19**	8	11	12	16	123
MiniZinc-PDB	6	12	**15**	12	**13**	18	9	11	11	16	123
Scorpion	1	**12**	**14**	12	**13**	0	**10**	13	**15**	14	104
SymBiDir	**14**	9	12	11	**13**	**19**	8	4	6	**18**	114
Complementary1	10	11	14	12	12	18	8	11	11	16	123
Complementary2	6	**12**	13	12	**13**	18	8	**14**	12	16	124
Oracle	14	12	15	15	13	19	10	14	15	18	143

Table 3.3 Coverage of PDB-type planners on the 2018 International Planning Competition for cost-optimal planning

We deduce that with GreedyPDB, we have a simple automated pattern selection strategy in a PDB planning system that is competitive with the state-of-the-art. The most surprising results can be seen in the difference between PlanningPDB and MiniZincPDB, both being almost the same planner, except for the method in which they do the bin packing subroutine. The results are very encouraging for the work in how to automatically select a combination of patterns to be used in a heuristic.

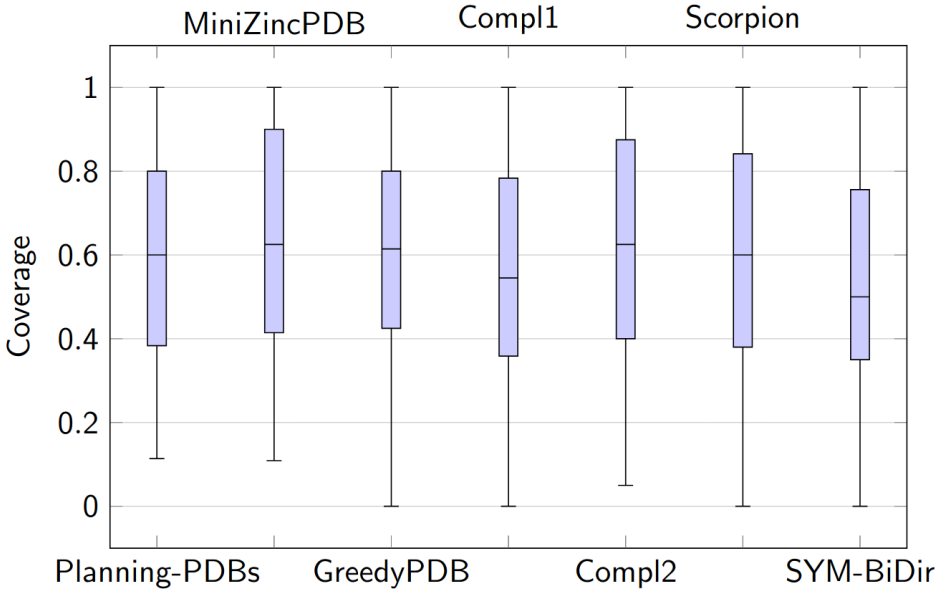


Fig. 3.4 Boxplot results of the normalised coverage (instances solved divided by the total number of problems in a domain set) across each domain in the suite of benchmarks across all IPC.

Figure 3.4 displays the distribution of normalised coverage (number of problems solved divided by the total number of problems in a domain set) the planners achieved on each of the domains. This gives us more insights about how domain independent each approach is, compared to the simple coverage score (total number of problems solved across all domain sets). The first interesting result is that the methods based on bin packing for pattern selection and Complementary 2, were the only planners that solved problems in all the domain sets. Planning-PDBs and MiniZincPDB were especially good, solving at least one in ten problems attempted per domain set – showing that the approach of maximising the use of memory when creating a PDB will always result in useful heuristic values.

Another aspect that results from Figure 3.4 is the distribution of coverage that each planner has. Here, MiniZincPDB and Complementary2 can be seen as having large differences between the first and third quartile of results, with GreedyPDB being the narrowest. These results, together with the fact that all three have very similar medians, makes us conclude that GreedyPDB is the best planner for domain independent planning, as it will reliably solve most instances across any domain.

When comparing MiniZincPDB with Planning-PDBs, the results show that even though the bin packing solution does not have any knowledge of the patterns it is fitting into the memory limits, there is merit in trying to maximise the capacity of the PDBs. This together with the GA optimisation will produce better PDBs than by using greedily constructed patterns.

In the 2018 benchmark, likely the most challenging one featuring a wide range of expressive application domain models, GreedyPDB, would have actually won the competition (Table 3.3). This indicates that for several planning problems, the best option is to keep growing one PDB with the greedy pattern selector, and compare and merge the results with a PDB collection based on bin packing. When looking at the combined benchmarks of the last three competitions, MiniZincPDB gives the best results, proving that improving the way patterns are combined gives a huge gain coverage (while comparing with PlanningPDB).

3.7 Related Work

Search algorithms have been found to be effective for many tasks, and a lot of prior work from research in mathematical graph problems have managed to kickstart the subject of search in the late 1950's, early 1960's. With the field expansion to include heuristics to guide the search process, the field of heuristics has been one of the most reasearched subjects in Artificial

Ingelligence's relatively short history. For AI Planning, many types of heuristics have been created, with most being specialised on specific domains, but few have shown generally good results across in domain independant evaluation. Pattern Databases heuristics have been, for the past decade, the most reliable approach for domain independent planning.

Pattern Databases have become very popular since the 2018 International Planning Competition showed that top five planners employed the heuristic in their solver. However, the topic has been vastly researched prior to this competition, a lot of work going in the automated creation of a PDB, with the best known being the iPDB of Haslum et al., [55] and the GA-PDB by Edelkamp [28]. The first performs a hill-climbing search in the space of possible pattern collections, while the other employs a bin-packing algorithm to create initial collections, that will be used as an initial population for a genetic algorithm. iPDB evaluates the patterns by selecting the one with the higher h-value in a selected sample set of states, while the GA of the GA-PDB uses the average heuristic value as its fitness function.

Another two approaches related to our work is Gamer [82] and CPC [43]. The first is in the search of only one best PDB, starting with all the goal variables, and adds the one that it will increase the average heuristic value. CPC is a *revolution* of the GA-PDB approach, aiming to create pattern collections with PDBs that are complementary to each other. It also employs a GA and its evaluation is based on Stratified Sampling.

A big variety of work has been included in PDBs, which might be one of the reasons why there has been such a big amount of research done in the subject. While most of heuristic search research seems to be reaching saturation on current search algorithms, the subject of PDBs seems to still have space for more cross pollination with other subjects, mostly from the Operation Research field.

3.8 Conclusion

In this chapter, we investigated the first objective of this thesis – **O1** – by investigating methods of solving cost-optimal planning problems with Pattern Databases – **C1**. We studied the pattern selection problem for generating better heuristics. We compared the pattern selection methods that generate many smaller PDBs, done via bin packing using greedy algorithm or CP solvers, with solutions that generate one best pattern with the Greedy Selection method – **C2**.

The best results were seen to be when the two solutions were combined, with the Greedy-PDB having state-of-the-art results across all modern domains, while MiniZincPDB proving the importance of how the bin packing problem for pattern solutions is solved. This shows that the more memory used for the heuristic will result in better results (coverage and normalised coverage).

Our solutions greatly simplified the previous best pattern selections mechanism, the CPC heuristic, by completely removing the sampling mechanism from the finding pattern generation mechanism. Our results have also shown that optimisation via Genetic Algorithms are still necessary and improve coverage in all cases.

In the following chapter, we shall move our focus to objective **O2**, by reviewing the results of the 2018 Deterministic International Planning Competition, investigating what the results tell us about the state-of-art in cost optimal classical planning and seeing how symbolic PDB planners performed. This will help us propose new definitions of portofolio-planning and a new metric for measuring domain-independent planning.

Chapter 4

Cost-Optimal Track of the Deterministic IPC18

In this chapter, we will solve the second objective – **O2** – of this thesis, by reviewing and investigating the results from the most recent cost-optimal track of the deterministic International Planning Competition (IPC) that took place in 2018, which was part of the International Conference on Automated Planning and Scheduling (ICAPS). We will look at the competition as a whole, and focus on the topic of *portfolio-planners* the new tasks that they are trying to solve and their inclusion in a domain-independent planning competition.

We will describe the best performing single-planner approaches, namely Symbolic Planning, Pattern Databases and Cost Partitioning, and investigate how they achieved these results and compared to each other. This section is included, not to debate the results of the competiton or how it was organised, but to evaluate the new techniques that worked best in the competition, have a broad view of the domain-independant planning field, and how some of the teachings from competiton could be used in the broader field of AI and Robotics. Work presented in this chapter has been partly published in [101, 33, 93, 42].

The novelty of this chapter includes the qualitative review of the competition, and the proposal of a new metric to evaluate the domain independence of a planner. All the planners discussed here are available on the competitions website.

4.1 International Planning Competition

In the field of action planning, there is a common understanding that planning competitions run on a set of unknown and partly new benchmark problems, which have helped advance planning technology.

The IPC had its inaugural edition in 1998, held at the International Conference on Artificial Intelligence Planning Systems (AIPS). Over the years it has led to considerable progress in the development of new planners, the creation of a large pool of planning benchmarks for all versions of planning formalisms and it has built an identity synonymous with the state-of-the-art for planning in any of its forms (e.g. in 2018 we had Classical, Probabilistic and Temporal tracks).

At the beginning, events were held every two years, as planning research in the modern sense was developing fast, but recently, as the benchmark sets available were larger and more planners were broadly available for inspection, advances have slowed down. Competitions are now organized every 3-4 years, giving time for researchers to advance the field and implement any new idea.

Participants have been confronted with an increasing set of challenges including extended expressiveness of Planning Domain Description Language (PDDL), involved planning task metrics, inherent problem complexity, and problems that have grown in their complexity and sizes (of predicates and operators). While there are several tracks, from temporal to probabilistic and GPU-based, the one that has received the most submissions has been the deterministic IPC with its cost-optimal track.

4.1.1 Importance

The IPC has brought a lot of benefits for the subject as a whole, first and foremost with PDDL, the high-level modelling language that has now become an informal standard input for most planners. PDDL was used from the first edition, bringing all new versions and features for one of the subsequent editions. As all the domains and problems are formulated using this modelling language, almost all modern planners are built now to support one of the versions of PDDL and more recently RDDL (Relational Dynamic Influence Diagram Language) for Probabilistic Planning [126].

Continuing on the topic of benchmarks, each edition published either completely new or reinterpretations of domains with new problems, an increasing number and diversity of available benchmarks for the planning community. This gives planner developers a more complete way of evaluating their systems.

Finally, competitions in any field bring together any community, and it manages to evolve a field. Comparing in a closed environment a vast number of planners, each approaching problems in a different way, has the benefit of putting each method head-to-head without bias. As the benchmarks are not known prior to planner submission, developers of said systems need to focus on creating domain-independent planners, suited for any possible domain.

4.1.2 Planning Evolution

After each IPC, certain techniques have risen as the *state-of-the-art*. In the past, heuristic search was often the best approach, and certain heuristics were highly successful [63]. Symbolic search has also had success with SymBA*, a symbolic bidirectional planner [32].

Each winner of the competition has shown the planning community which combination of technique and domain works especially well. Most of the best performing planners have been awarded more attention in the following years, bringing forth their ideas in the community. Also, each well performing planner in the competition has led the organizer of the following competition to make their benchmark set harder for those techniques. This has made the community now have a very diverse set of problems in which we can see how well each planner performs.

Portfolio planning is a technique that tries to match the best planner for the domain/problem that it has to solve. Work done by Sievers et al. [131] has shown how grounded problems can be classified as to give the *better suited* planner the problem to solve. Following this work, *portfolio-planning* has become a viable solution to cost-optimal planning and has resulted in a change of paradigm during the 2018 Deterministic IPC, which we shall go in more detail over the following sections.

	Agri-cola	Cal dera	Data Net Network	Nuri-kabe	Organic Synthesis	Petri Net Alignment	Sett-lers	Snake	Spider	Termes	Σ
Delfi1	12	13	13	12	13	20	9	11	11	12	126
Complementary2	6	12	12	12	13	18	9	14	12	16	124
Complementary1	10	11	14	13	13	17	8	11	11	16	124
Planning-PDBs	6	12	14	11	13	18	8	13	11	16	122
SymbBiDir	15	10	13	11	13	19	8	4	7	18	118
Scorpion	2	12	14	13	13	0	10	14	17	14	109
Delfi2	11	11	13	11	13	9	8	7	7	15	105
FDMS2	14	12	9	12	13	2	8	11	11	12	104
FDMS1	9	12	10	12	13	2	9	11	11	12	101
DecStar	0	8	14	11	14	8	8	11	13	12	99
Metis1	0	13	12	12	14	9	9	7	11	6	93
MSP	7	8	13	8	12	10	0	11	6	16	91
Metis2	0	15	12	12	14	9	0	7	12	6	87
ExplBlind	0	8	7	11	10	7	8	12	11	10	84
Symple-2	1	8	9	7	13	2	0	0	5	13	58
Symple-1	0	8	9	8	13	2	0	0	4	13	57
maplan-2	2	2	9	0	6	0	0	14	1	12	46
maplan-1	0	2	12	0	6	0	0	7	10	6	43

Table 4.1 The results of the Cost-Optimal track from the Deterministic IPC 2018. Planners are measured based on the coverage (i.e, in the instances of tasks solved per domains) across all the new domains, each consisting of 20 instances increasing in difficulty.

4.2 The Results of the 2018 Deterministic IPC

The outcome of the optimal track in the most recent 2018 International Planning is revisited in Table 4.1, with a description of all the planners available in the planner abstracts that can be found in the competition booklet at *https://ipc2018-classical.bitbucket.io/planners*.

While most planners present one sole planning technology, the winning planner Delfi [131] is a *portfolio-planner*, which uses a mixture of different planning technique and technologies, and operates as follows: given a problem task, it selects a planner (from a set of planners) to use when solving it, and is based on a classifier which was trained on a manually chosen set of known planning benchmark instances.

In its set of planners, 16 were contained as part of the Fast Downward planning framework, including at least one that used Pattern Databases with their implementation of the Canonical PDB [56]. The most effective approach chosen by this classifier, however, was the symbolic search planning system, which won the precursor 2014 IPC competition. Moreover, in the only domain, where Delfi scored overall clear best, it called this planner.

From the point of solving a planning task, portfolio-planners do not bring any novel contributions. Instead, they focus on finding better methods to map planning tasks to planners

that could solve them. This is done by transforming the domain/problem pair into a bit-vector image and classifying which planner has best solved such an image in the training set (consisting of all IPC benchmarks previously used).

As such, in Delfi, one planner is called per instance. In terms of the potential of different planning approaches available to Delfi, the IPC outcome with a lead of two more being solved ($\approx 1\%$ of 200 benchmark problems) is quite small, showing that cost-optimal planning is tough, even for portfolios. A change in one domain might have resulted a different outcome.

While portfolio planning was in alignment with the rules of the competition, one underlying issue is some participating planners avoided using portfolio technology, presumably in favour of getting a clearer picture on what technology is currently leading.

Facing the outcome of the competition and the different type of contributions available in portfolio and non-portfolio planners, people interested in planning especially outside to the core planning community have to be warned not to derive wrong scientific conclusions by only looking at the outcome. Competition results always have to be dealt with care, and a clear distinction should be created between solving domain-independent planning tasks via one method, or via task-mapping to pre-existing planners.

In the remainder of this chapter, we will try to create a clearer picture on what is currently the best approach to domain-independent cost-optimal planning, according to the results of the deterministic IPC 2018. We will also discuss whether portfolios help to push or blur the outcome of a competition.

4.3 Symbolic Search and Pattern Databases

The IPC 2018 planner *Symbolic-Bidirectional* (SymbBiDir) was used as a baseline planning technology. It includes no lower bound at all and, thus, relies on so-called *blind* search (i.e., a search with no heuristic search guidance). As actions carry cost, instead of a breadth-first exploration, this induces a cost-first traversal of the state-space graph.

As described in the background chapter, the core difference between *symbolic planning* and explicit-state space planning is the use of binary decision diagrams (BDDs) to represent state sets in the search compactly [9]. As actions can also be represented in the form of BDDs encoding the transition relation, it is possible to progress and regress planning state in this succinct functional state set representation, to perform forward and backward exploration in an

operation called *relational product* [16]. A first A*-type algorithm for BDD-based heuristic search was proposed by Edelkamp in [34].

One observation is that the memory savings obtained via a more compact representation in a BDD often lead to significant savings in processing time, and thus more search time. The gain of a symbolic representation in IPC 2018 is amplified, when comparing the performance gap of *SymBiDir* with the other baseline planner *ExplicitBlind*. The two baseline planners are not executing the same exploration, due to the fact that coding regression search is not immediate for the usually given partial goal representation; so that the latter conducts a forward state-space traversal only.

SymBiDir executes bidirectional cost-based search, much in the sense of the bidirectional application of Dijkstra's single-source shortest path algorithms [23], taking into account that the optimal solution might not be established on the first meeting of both search frontiers. As the BDDs represent state sets, recursive solution construction is needed for extracting optimal plans. Aspects such as a partitioned computation of successors (called the *image*), variable ordering based, as well as the inclusion of invariant constraints to rule out illegal and dead-end states are essential factors for improving the exploration efficiency [135].

The performance results of *SymBiDir* revealed that in only two of the ten domains – Snake and Spider – it performed poorly, but it was the best performing across the other eight. This indicates the power of symbolic state-space representation and exploration, suggesting that at least across the entire IPC 2018 benchmark set, the vast amount of research done to refine heuristic search for AI planning did not lead to the best results.

4.3.1 Planners using PDB Heuristics

The result of the 2018 competition also show that, planning using PDBs as a heuristic [25] seems to be one of the few exceptions to the dominance of symbolic blind search. On the test bed of IPC 2018 the combination of PDBs and symbolic search in the planners Complementary (1 and 2) and Planning-PDB outperformed *SymBiDir*.

The Complementary planners are based on the results of Franco et al. [43], while the latter was based on our work on bin-packing algorithms for the pattern selection problem, shown in the previous chapter. Besides a major rewrite, one new feature of these new planners is that the forward search is in fact explicit-state, while only the backward traversal is based on symbolic.

Of course, many heuristics besides PDBs are still worth further investigation. If the Snake and Spider domains were removed from the competition, it would be difficult to draw a conclusion on different types of heuristics from the IPC-18 results.

There is no free lunch. But the overall performance of bidirectional symbolic search is surprisingly good, while not using any heuristic. There is still more work to be done in investigating which aspect of this approach helps the most in achieving a good performance, checking if it is the bidirectional approach or the symbolic search, but we expect one would need both.

SymbBidir performs much better in the Agricola domain than all other planners in the competition, but a PDB-approach seems to hurt symbolic search, resulting in 5 to 9 instances where SymbBiDir succeeds while symbolic PDBs fail to find a solution. This has been identified by us, and we overcome it by using *perimeter PDB* [38].

Regarding the poor performance in Spider and Snake, the reasons seems to be due to the BDD format *exploding*, which relate to the subtle ordering problem of dependent BDD variables in grids. This issue has been analyzed and proven to be crucial for representing the goal to the ConnectFour problem as a BDD [31], and might be detected fully automatically.

4.3.2 Scorpion Planner and Cost Partitioning

While the *Scorpion* planner, which also uses PDBs to inform its search mechanism, performs slightly worse to *SymBiDir*, the symbolic benchmark in IPC 2018 domains, it showed distinguished performance and scored best in half of the domains. Further investigations illustrate that it performs much better across all IPC benchmarks, i.e., including those from previous competitions [128, 129]. It also has to be added that part of the success of Scorpion is due to Cartesian abstractions combined with a counterexample abstraction refinement (CEGAR) approach [15].

As both PDBs and CEGAR are based on state-space abstractions, one could argue that they would reach heuristic estimates of a similar type, via different routes, but more work will need to be done to prove this statement. While PDBs and Cartesian abstractions are an important part of Scorpion, much of its strength is in the sophisticated method to *combine* these abstraction heuristics. While the competing symbolic PDB planners mainly use 0/1 cost partitioning, Scorpion uses saturated cost-partitioning, which can be attributed to the fact that

symbolic search has not yet been able to adapt this technique for splitting the cost across different planners in such a fine way.

In the past decade, symbolic search has great results for tasks which have a large search tree, being able to compress it into a much more efficient BDD (or related) format. However, the results shown by the Scorpion planner proves that there are still domains, in which the search tree has a higher depth rather than width, where it maintains an advantage. This shows that there is space for a future portfolio planner that is tasked with identifying which task is better suited for symbolic or explicit states.

Again, all or most planners, even the ones that did not have the best overall coverage, had some positive results (e.g. being among the planners that solve most instances in some individual domains). Even planners at the bottom have some cases where they perform among the best (e.g. ma-plan in the Snake domain). As such, we identify that portfolio planning can have a huge impact in domain-independent planning, and we will continue by describing and analysing what it specifically is in more detail.

4.4 Portfolio Planning

Portfolio-planners created based on existing planning search techniques are a recurring pattern in many of the other tracks in the IPC, from deterministic-satisficing to the temporal track. They range from restarting strategies over learning classifiers, to scheduling time slices to existing planners.

The organisers had decided to allow such systems in the competition, due to the fact that it is difficult to accurately define what is and what is not a portfolio-planner. One example of such a system is the LAMA system [122], one previous IPC winner, which might have to be considered a portfolio, because it runs different planners one after the other.

This said, there are certainly many interesting aspects to be learned from portfolios on a per-domain or even per-instance base. Proper portfolio designs with a close-to-optimal choice of planners as in *Delfi* is a research area on its own.

One way to limit the impact of portfolios in the competition is what the organisers of the Sparkle planning competition (for more information about the Sparkle competition, see http://ada.liacs.nl/events/sparkle-planning-19), where planners are evaluated based on how well they do on individual instances/domains, rather than achieving a good average score.

This approach, however, comes with some issues as well. In particular, the score of a planner completely depends on which other planners are submitted to the competition. Henceforth, if someone submits a version of your planner that works only slightly better, they could get 0 points. One has to wait for the results to see how the approach materializes. Unfortunately, the organizers of this competition are only running the agile track, without any track for cost-optimal plans. In complexity terms, however, optimality is known to be of crucial importance. For example, finding any plan for many planning benchmarks (such as Blocksworld, Logistics, Sliding-Tile Puzzle) is polynomial. In the case of *satisficing* planning, where only plan existence is requested, this tends to be tractable, whereas the corresponding optimisations are often provably hard [132, 110, 62].

The emerging set of *portfolio-planners* and the difficulty of excluding them from future competitions may be seen as a side-effect of the requirement of releasing source code for the planners, as it becomes easier to bundle planners into one code base. Of course, public access to the source code is not a strict necessity for these type of planners.

For some planning researchers, the core issue and concern of portfolio planning is that other researchers use their code, and not so much that the participating planner is a portfolio. This led the organisers to think about creating a rule against using code from differen research groups, but that would have led to the exclusion of most planners, as they were based on Fast Downward planner framework.

The solution introduced in SAT competitions was having a license that prohibits the use of the code in other tools. This could be an option for future IPCs, but then the authors of the planners would have to implement such a change before the next competition. In case of planners based on Fast Downward, this, however, is also problematic because such a clause would be hardly compatible with the license of the framework.

4.4.1 Defining Portfolio Planning

As stated prior, it is difficult to create a definition to distinguish a portfolio from a non-portfolio planner in a formal manner. One may try to start with the following criteria.

Definition 25 (Portfolio-planner) *A* portfolio-planner *selects, invokes, and possibly terminates different existing planners, based on a trained or hard-coded decision procedure.*

This definition may not be a perfect discriminator, as one might be able to transform a portfolio into a non-portfolio without changing the performance by much: just moving the decision procedure further down the line.

It also does not cover planners that use the maximum of several heuristics in an A* search, approach which could also be considered a portfolio of heuristics. This could also lead to no contribution except for the selection procedure between the heuristics. At the end, the question remains on when a planner is a novel contribution.

Another possible definition for identifying portfolio planning is the following:

Definition 26 (Non-Portfolio Planner) *A* non-portfolio planner *is a single core planning technology, which invokes a plan search in one state space.*

But what about traversing state-space abstractions, which are needed to compute heuristic estimates? Clearly, as highlighted by the IPC organisers, defining portfolios turns out to be intrinsically difficult. There are planners that are clearly portfolios, there are planners that are clearly not, and there is a larger grey area in between.

According to a definition, FF [69] should not be judged as a portfolio planner. It searches one state space with one heuristic. But FF switches from enforced hill-climbing to best-first search, based on some progress measure. This alone should not classify it as a portfolio approach.

LAMA runs a greedy-search based on h^{FF} and a landmark heuristic (three techniques developed by different authors) and then several weighted A* searches (a different planner and an algorithm also developed by other authors), leading the approach in a grey area between portfolio and non-portfolio planning. If it runs three independent searches in parallel, then this may be interpreted as a portfolio technology, but the interconnection of the search is more subtle. LAMA had additional algorithmic contributions on how to move back and forth across the states in the different priority queues. If LAMA continues searching the same search space, this is a sign of a non-portfolio, as it is not utilising the executable of a different already existing planner.

It is, however, abundantly clear that Delfi is a portfolio planner, as stated by the developers behind it. Delfi utilises two executables, SymBA*, and 16 parameters of selecting planners in the Fast Downward framework. It has a decision procedure trained on a set of manual

selected planning tasks. Note that in this setting, we do not count the learning as running, but as programming time.

The performance overhead of portfolio designs can be small. The often criticized effect is that frequently more than 99.9% of the actual running time of a portfolio planner is exclusively spent on executing previously created planners. This is a probably unwanted aspect, which can lead the competition to not attract new planners and contributions to the field of task planning. Of course, the size of a contribution must not necessarily be taken into direct correspondence with the profiled time that was spent executing the code added. All the results and logs created by the planners in the 2018 planners is publically accessible, leading to anyone deciding if those planners use a portfolio approach or not.

4.4.2 Delfi Planners

Based on our interpretation of the competition's logs, there were at least two different portfolio planners in IPC 2018: *Delfi1* and *Delfi2*, where Delfi1 performed much better and so that in the following section we will concentrate only on it. We will use *Delfi* for identifying the general idea presented in both. There is published work from the planner authors in the IPC booklet that explain the architecture and the machine learning approach of using deep neural nets in more detail, so we will focus on the main aspects.

The idea behind this portfolio is to train a classifier on the performance curves of known planning benchmark problems, provided as input images. We had some problems to reproduce the results on our machine, but look at the competition results for reference. The planners being invoked by Delfi in the IPC 2018 are shown in Table 4.2. Note that Delfi combines the heuristics listed with symmetry and partial-order reduction.

4.4.3 Domain-Independent Planning

The story on portfolios will go on. Portfolio-planners have already added the planners that took part in the IPC from 2018, resulting into a very difficult task to finding better approaches other than by creating radically new planners.

While portfolios have dominated some tracks in the IPC in the optimal track, the winner of each edition until (and including) IPC 2014 was not a portfolio. As seen in IPC 2018, there were several planners that got a very close performance to Delfi. Also, some other portfolios

Approach	Used	Successfully
SymBA*	110	73
LM cut	64	37
Merge&Shrink	47	20
Canonical PDB	17	13
Blind search	2	2
Total	240	147

Table 4.2 Planners chosen by the Portfolio Delfi1 based on analysing the log files of IPC 2018. Only main planner technologies are mentioned, many more parameters apply to the actual invocation of the code. Note that the number of problems being solved is slightly higher than in the competition outcome, as there were some reformulations of the same problem, where the planner was run too.

participated. Delfi2 and other portfolios (MSP and DecStar could be considered portfolios as well) were behind many non-portfolio planners. Overall, the results do not show the dominance of portfolio-planners in general.

As stated before, research into portfolio-planners has its own contribution and should not be ignored. Finding good combinations of different techniques is not a trivial task, and there should be a focus on this, with future competitors needing to focus on such problems (e.g., how much better is this combination rather than just running all n components for 1 n-th of the time).

Portfolios like Delfi based on Neural-Networks should be included in a different learning track, where the focus should be of feature-extraction and training the best classifiers for choosing the best planner-task pair.

Based on the current results, there is still much work to be done in this field, as Delfi was marginally better than the single-approach ones, which we think is the main conclusion that can be drawn from the 2018 cost-optimal deterministic IPC.

4.5 Measuring Cost-Optimal Planning

In this section we will be discussing different ways of comparing planners, and see how they differ from each other. We have tested five planners, Complementary 1 and 2, Planning-PDBs, Scorpion and Symbolic-Bidirectional on 69 domains, all the benchmarks from the previous three competitions and a subset of the domains from before 2011.

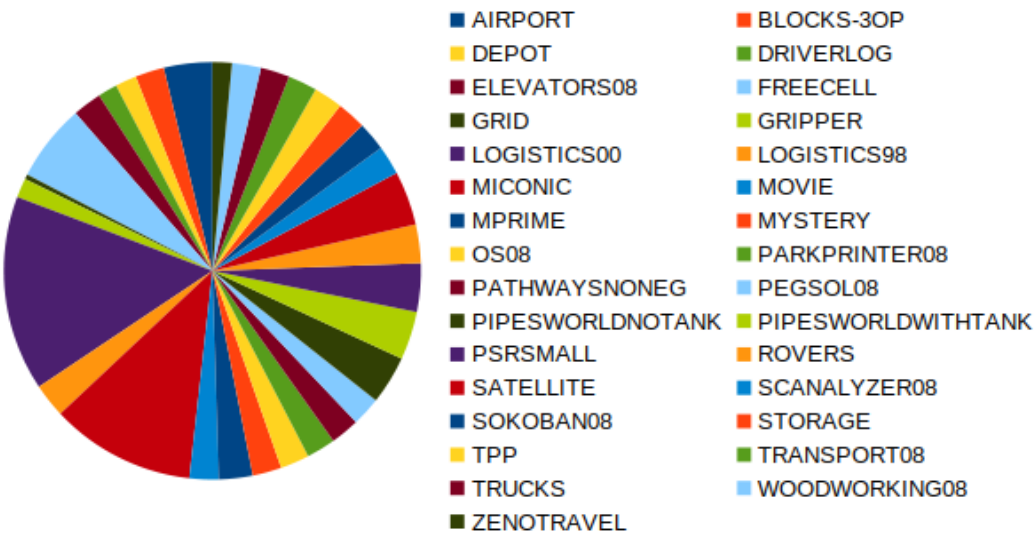


Fig. 4.1 Size of domains from our pre-2011 benchmark

4.5.1 Coverage

As stated in the first section, the current way of comparing cost-optimal classical planning is by measuring the coverage of a planner (i.e. how many problems a planner can solve on a set of problems). Each problem solved is counted as a point towards that planner and at the end we compare the tally of each planner, the one with the most being the winner.

This metric is used both in competitions and in published papers when measuring the performance of a new method. However, this metric can become domain dependant if the number of problem instances is not uniform over all the domains. In our pre-2011 set of problems, made out of 31 domains, we can see that some domains are a lot more important than others when using this approach (seen in figure 4.1).

For the benchmarks from 2011 and 2018, the domains were kept at a uniform size of 20 instances each. In this case, there is no need to normalize the results, but when using benchmark sets like 2014 (most had 20 instances, with three different) and the pre-2011 we used (from 5 to 202, with most having 30 instances), the change in domain sizes requires a change in the evaluation metric.

	Problems Solved	Coverage	Normalized Coverage
Planning-PDBs	1122	54.17%	59.42%
Complementary1	1099	53.06%	57.60%
Complementary2	1164	56.15%	**62.08%**
Scorpion	**1208**	**58.32%**	60.11%
Sym-BiDir	1053	50.84%	55.46%

Table 4.3 Overall results as number of problems solved, coverage and normalized coverage.

	Pre 2011	Coverage	Normalized Coverage	IPC11	Coverage (also Normalized)	IPC14	Coverage	Normalized Coverage	IPC18	Coverage (also Normalized)
Planning-PDBs	678	50.78%	55.88%	190	67.85%	131	51.17%	53.48%	123	61.5%
Complementary1	680	50.93%	55.95%	185	66.07%	111	43.35%	46.15%	123	61.5%
Complementary2	686	51.38%	56.95%	**198**	**70.7%**	**155**	**60.54%**	**61.99%**	**124**	**62%**
Scorpion	**785**	**58.80%**	**60.20%**	190	67.85%	118	46.09%	48.77%	104	52%
Sym-BiDir	647	48.45%	53.46%	174	62.14%	129	50.39%	52.97%	114	57%

Table 4.4 Results of the five planners on the pre-2011, IPC11, IPC14 and IPC 18 benchmarks. For each benchmark we have the number of problems solved, coverage and normalized coverage (where needed).

4.5.2 Normalized Domain Coverage

For cases like this, we normalise the domain coverage, and then get the average for each planner. By doing this, we first see how much of a domain a planner can solve, and then by averaging we get a better metric for overall domain-independent performance of a planner.

We can see the value of such a metric in table 4.3, where, even though Scorpion solves the most problems out of the total of 2071 we tested on, the normalized coverage is worse than that of Complementary2 (62.08% to 60.11%).

By looking at Figure 4.2 of the normalized per-domain coverage of each planner, we can see an even clearer picture. The only two planners to solve problems on all the 69 domains are Planning-PDBs and Complementary2. Also, Planning-PDBs is a lot closer to Scorpion than what the number of instances solved would imply (1208 to 1122).

Another way of measuring the performance when having a normalized coverage, would be by getting the median value for each planner. In our test cases, we find that Complementary2 has the best with 65%, with Scorpion and Planning-PDBs following (both have 60%). Complementary1 and Symbolic-Bidirectional finish the top with 55% and 50%.

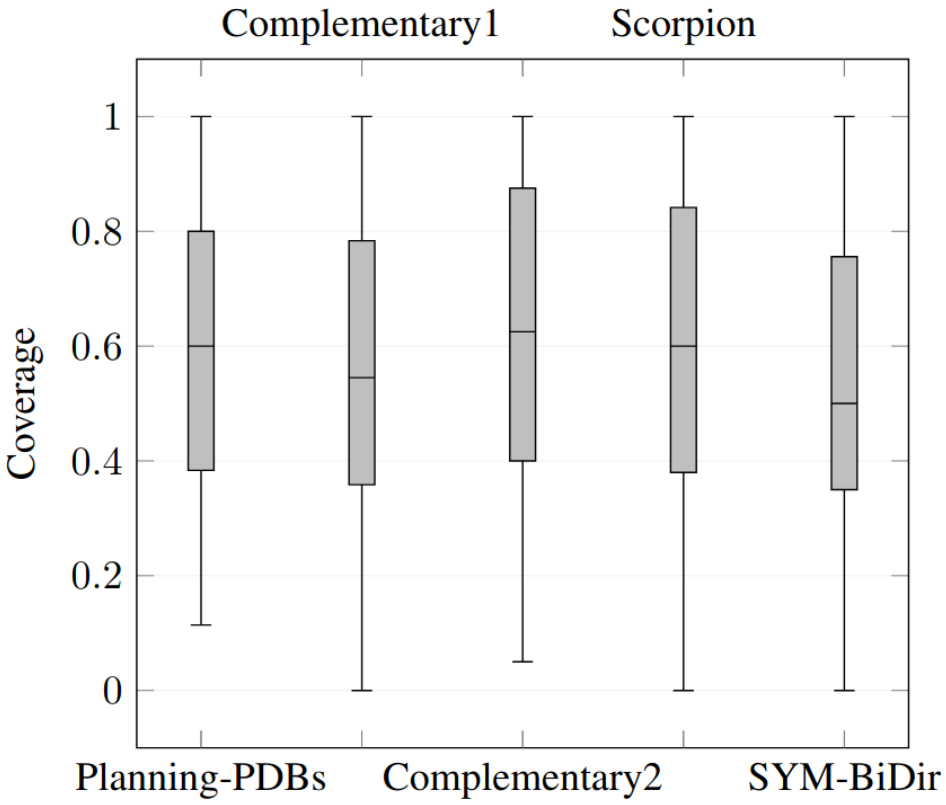


Fig. 4.2 This figure shows a boxplot of all the normalised results across each domain in the suite of benchmarks available from all previous IPCs. As domains have different numbers of problems, basic coverage will not be a clear indication of how well the planners perform across several domains, but with a boxplot it is possible to get more information in a more concise representation. One such point is that Planning-PDBs and Complementary2 are the only planners that were able to solve problems across all domains, or that while the Complementary1 and SYM-BiDir planners have similar median values, the confidence box of the latter is not as high as Complementary1.

4.6 Conclusion

This chapter focused on Objective **O2** of this thesis, investigating the current leading cost-optimal planning solutions based on the IPC results, with a discussion focused on portfolio-planning and if they are pushing or blurring the outcome of the competition – **C3**. There are indications, but not one definite conclusion to which approach appeared best on that set of domains. However, the need for a separation between planning and portfolio mapping in domain-independent competitions such as the IPC has emerged in it.

The IPC has always been a competition where the best mix of scientific and engineering skill wins. It has never been just one good idea that won the competition, but also the skill to implement it efficiently and showing to the whole field without any barrier. From this edition, we can conclude that:

- Pattern Databases appear to offer the best results when using only one technique;
- Cost-partitioning methods are extremely important when guiding search with several heuristics;
- Planning task mapping to planners has emerged as a valid approach to solving cost-optimal planning, and will offer the best solutions in domain-indenpendent scenarios.

After previous editions of IPCs, there had a discussion on domain-independence control rules in TL- and TAL-Plan [2, 90], as well as relating to the effect of hand-coded selection in planners like SGPlan [142], with complaints on handwritten domain-dependent branching inside the planners' code. After the 2018 edition, the main topic was portfolio planning and the problem of identifying individual planner contributions.

The International Planning Competition 2018, as any of the previous editions, aims at pushing the field of task planning. It set up and executed a well-designed externally controlled experiment, aimed at offering insights regarding the performance of planners, falsified and strengthened hypotheses on essential components, and compared different technologies on a common rule set, same architecture, and an agreed input formalism. It awarded scientific prizes and provided opportunities for upcoming publications. The evaluation it does offers an increased visibility than those seen in conference and journal papers, increasing reach and impact of the underlying research of each planner participating. The results have often been surprising, when compared to the wisdom taken from existing publications. Of course, every competition is limited in what it can prove, but its scientific impact should not be underestimated.

Our final contribution of this chapter – **C4** – was to propose a better metric and visualisation for analysing the domain-independence of a planner: *normalised-coverage*. The current set of planning benchmarks induces biases towards the domains with more instances available, resulting in approaches like saturated-cost partitioning PDBs (Scorpion) solving the most instances across all domains due to being better in a handful of older domains (e.g. Logistics00).

The following chapter will investigate the field of planning under uncertainty for robotic agents, and how Plan Libraries, an idea inspired by Belief-Desire-Intention agents, can be used

to make deterministic planning a fast and robust solution. This will lead to reaching our final Objective – **O3** – and resulting into contributions **C5** and **C6** of this thesis.

Chapter 5

AI Planning with Robotics by using Plan Libraries

This chapter includes our work for achieving the third objective of this thesis – **O3**. We will investigate how agent reasoning can be aided by storing previous plans and executions in the memory of the agent. This approach takes inspiration from the Belief-Desire-Intention agent paradigm, in which most approaches typically make use of a *Plan Library* that stores abstract representations of the agents possible behaviours.

Our approach is intended to be added on top of an agent's higher-level reasoning stack, which reasons about their own actions by using a PDDL planner. This leads to making their previously computed plans reusable, leading to effective robotic executions while operating in dynamic environments, with many other agents or humans may operate and affect the environment. We envision that this will find use in robotic environments, using this as our domain for evaluation.

We will begin this section by going over the motivation that informed our work, showing the need for the agents reasoning process to be prompt in their plan creation process. We follow this by describing the theoretical ideas behind a Plan Library to be added in the reasoning of an agent using AI Planning for their task planning.

Furthermore, we finish this chapter by describing our implementation of a Plan Library as a ROS node added to the ROSPlan framework and reasoning cycle. In conclusion, we will evaluate our extended ROSPlan system with the generic one, seeing how being able to reuse plans affects execution times and memory usage.

Some of the work presented in this chapter has been published in [100].

5.1 Motivation

We began this thesis by describing the manner in which AI has been introduced in different fields, with many areas having already integrated AI solutions in their workflows. However, there are still many more areas that have shown the need for robotic agents to be added in the workforce.

This need is especially visible in areas that have been seeing a decrease in their human workforce, due to factors such as low wages, high impact on their mental and physical health, and also the intrinsic dangers their tasks could entail. Such fields extend from agriculture to mountain rescue, and have already seen research and real life integration of robots [144, 75, 96], albeit using systems that have limited autonomy and little decision-making capabilities.

5.1.1 Autonomy and Speed

In order for robots to become useful in real world environments, their reasoning process needs to combine autonomy with speed. The real world is an environment defined by its dynamic and ever-changing nature, with many humans-in-the-loop, increasing the uncertainty in the model.

Operating in such an environment leads to issues for both model and data-based agents, as it is difficult for a learning-based agent to learn or adapt in an environment that is different from that which it was trained. On the other hand, model-based approaches need a way to deal with uncertainty in the environment.

There are many tasks to be solved for model-based reasoning to become robust enough for a robot to use it. They vary from maintaining model integrity while executing a plan to the computational intractability of expressive models (which would allow agents to reason about models that closely resemble their environment) therefore trading speed for expressiveness.

Robots need autonomy in their decision-making in order for them to be useful in most tasks in which they need to replace humans. Many tasks need on-the-fly decisions, such as deciding if a fruit is ripe or a path is blocked, and robots need to be able to decide for themselves how to achieve their goals, regardless of the situation they are placed in [76]. This requires speedy

reasoning so that they can perform in dynamic environments where plans can become unusable if a robot takes too long to synthesise them.

One way to ensure that the robot has a high degree of autonomy is by reasoning directly about the state of the environment, using a *planner*. Such an approach trades speed for autonomy, because planning with a suitably expressive language — for example, ability to reason about the duration and cost of actions — is computationally expensive.

The robot's successful integration in a dynamic environment also depends on the speed of which planning is done (i.e.: the time from which the planning component receives a planning task until it submits the plan for execution). If the robot has a valid plan for a task, but the task is no longer consistent with the current state of the environment (due to the long time taken to create the plan), then it needs to re-plan, restarting the reasoning process.

5.2 Plan Libraries

Because of the issues in the speed of an agent's reasoning, approaches that make use of predefined plans have been developed. One influential family of approaches are those based on the Belief-Desire-Intention (BDI) paradigm [8]. Systems that are based on the BDI model include PRS [121] and Jason [7] which have a prescribed Plan Library, comprising a set of plan rules. Each plan rule consists of a *header*, which defines the situation where it is applicable, and a *sequence of actions* that will fulfil the robot's goals. The downside to this approach is that the robot is limited to the behaviours described in the Plan Library, and therefore has less autonomy than a robot that can compute its own plans.

Over the last decades, researchers have introduced task planning into BDI agents [97], where the main focus is planning when there is no available option in the Plan Library. In other words, planning is considered to be a last resort, only to be invoked if necessary.

Our solution is to take a complementary approach, starting with no plan library, carrying out task planning to achieve goals, and storing the plans that are generated, reusing them where that is possible.

All the changes from a BDI agent to our implementation of a Plan Library have originated from the difference in implementation of a PDDL planner from a BDI agent or by how the system is affected by having a much higher number of plans (due to them being included in the library in their grounded representation).

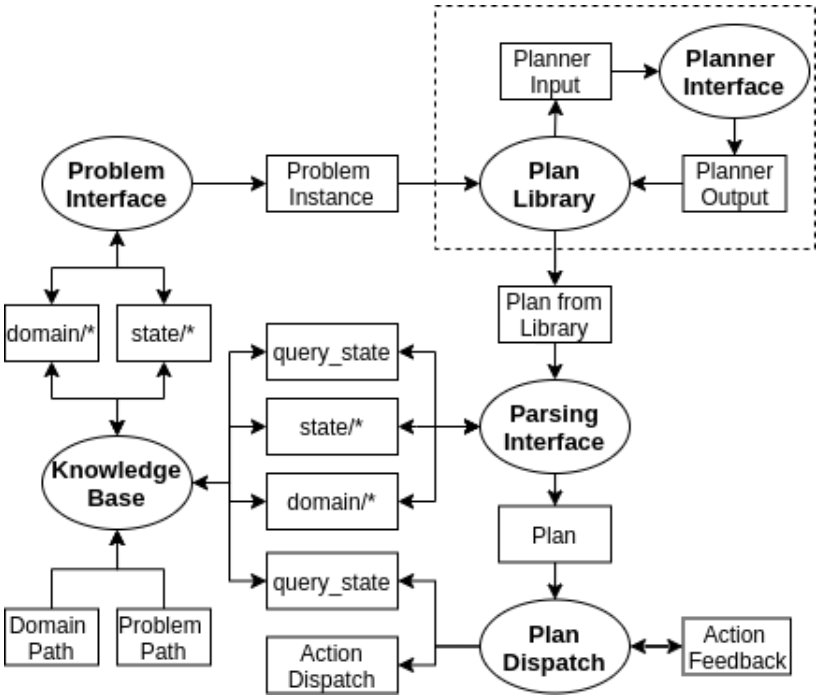


Fig. 5.1 Plan Library

5.2.1 Plan Library Node

The ROSPlan framework offers a standardized approach for integrating task planning into the Robot Operating System (ROS). The framework consists of five ROS nodes (processes that perform computations) and several topics (named buses that exchange messages between nodes): Knowledge Base, Problem Interface, Planner Interface, Parsing Interface and Plan Dispatcher.

The Plan Library ROS Node, identified in Figure 5.1 by the dotted box in the upper right, is a proxy for the Planner Interface from the default ROSPlan framework. Previously solved problems together with their plans (stored on the machine in YAML format), are loaded as a dictionary when the agent is initialising. If this is the first time the agent is operating in the environment, the dictionary will be empty[1].

[1] In the case that a differenent agent with the same capabilities had previously operated in the environment, it is possible for their plan library to be loaded onto the new agent.

5.2.2 Plan Storage

As a plan library is expected to work on one specific planning area/domain, in the current implementation we assume that the domain will not change between executions. Each problem file is parsed, receiving the initial and goal state from the PDDL file and added as individual queries for the plan library dictionary.

Each query element consists of *initial* and *goal* predicates, which will be used when searching in the plan library dictionary for previous solutions. For a query to be matched with an entry in the plan library, the following will need to be true:

- The initial states will need to match, having a corresponding predicate in the query element for each predicate in the plan library entry. Variable names can be ignored, as each element of the same class should have the same facilities.
- The query goal state should match or be a subset of that from the plan library element. This is due to goals in a PDDL planner being a partial state representation.

This approach is based on how Plan Libraries in BDI agents are constructed with each plan having a *header*, which states what scenarios that plan is available to be used. One difference is that in our case, it is necessary to include the *goal* condition (i.e. the desire of the agent) in the header of the element, for a quicker search in the Plan Library dictionary.

5.2.3 Plan Quality Metrics

In BDI agent design, there are usually no metrics involved for comparing how different Plan Library elements compare with each other, as the designer is expected to have created a proper model for the tasks that they need to solve. There has been work done by Xu et al. [143], that focused on creating a framework for modifying the Plan Library over the long term execution and run of an agent. For the library to be expandable and contractable, Xu et al. defined several performance metrics to evaluate a plan.

The metrics of interest to us are *execution count*, *success rate* and *replaceability degree*. They are inspired by Markovian Decision Processes (evaluating the quality of a state), and are helpful to identify how useful one plan is compared to the others.

The first two are trivial, as the measure of how many times has a plan been executed and how many of those times has it been completed successfully.

The latter metric, *replaceability degree*, keeps track of how many other plans in the library can be used for a specific task submitted to the agent, and averaged across all times, a plan has appeared as an option for solving a task.

All these three metrics are kept in the Plan Library dictionary with their respective entries, and are used to decide which of the applicable plans for a task should be used.

5.2.4 Domain Exploration

Exploration is necessary for the plan library to become useable and the metrics applicable, otherwise the agent will have a greedy behaviour and keep reusing the same small set of entries from the plan library. As such, we decided that the best approach for keeping the agent exploring is by using two thresholds when deciding which plans it should use:

1. If there are not enough (used defined threshold) plans to select from for a specific task, the planner should be invoked to search for a different plan by using different search parameters (e.g. different heuristic, different timeout, etc.);
2. From the plans applicable, select one for execution that has not reached the required threshold for *execution count*.

We decided to use a threshold exploration function as it will aim to have a larger and more tested plan library, and will result in better executions. If the agent is not exploring, then the agent will go over normal plan selection, which we will discuss in the next section.

5.3 Plan Selection

An agent that works based on a library of experience (i.e. previously generated and executed plans), will need to reason about picking from a pool of possible plans for a specific task. We explore options for doing this, one being *greedy-plan selection*, and the other being *best-plan selection*.

Both proposed methodologies are PDDL agnostic, accepting all planning languages ROSPlan is able to process and generate.

The planning cycle of ROSPlan is as follows: It will first load its domain and problem file into the the Knowledge Base, which will be parsed into the state of the task. The Problem

Interface node will create it into an instance (this could be skipped initially, as the Knowledge Base has access to the PDDL files, but in subsequent iterations/in the case of replanning being necessary, this is vital for parsing the state into a PDDL problem instance that can be used as input for a planner) and send it as an input for the planner. This will come with a solution and serve it to the parsing interface, which will enter the acting loop, dispatching each action iteratively to the agent. While this is done, the state will be queried to know if there are any changes from what it expects to happen if the plan is followed, to the state of the actual environment. If any changes compromise the plan, it will replan by serving the current state to the problem interface, starting the whole cycle again, until the initial goals are reached.

5.3.1 Greedy-Plan Selection

Algorithm 5 Greedy-Plan Selection

Input: A problem instance $\mathscr{P}$, a dictionary of previously solved planning tasks `PlanLib`
Output: A plan π

```
1: (init_state, goal) ← 𝒫;
2: plan_found ← False;
3: π ← empty;
4: for entry in PlanLib do
5:     if entry[init_state] matches init_state, and entry[goal] matches goal then
6:         π ← entry[π];
7:         plan_found ← True;
8:         break;
9:     end if
10: end for
11: if not plan_found then
12:     π ← CallPlannerInterface(𝒫);
13:     PlanLib.append(init_state, goal, π)
14: end if
15: return π
```

Algorithm 5 describes how the greedy plan selection operates once the Plan Library is loaded. When the node receives a planning task as a PDDL file from the Problem Interface, it parses it into a *query element*. Next, the node iterates over the Plan Library, matching its initial state and goal elements with those of the problem it needs to solve. If there is a match, the iteration is interrupted and the plan from that Plan Library element is sent to the Parsing Interface.

If no problem from the Plan Library is found to match, then the problem is sent to the planner via the Planner Interface node. In the case that it returns a solution to the problem, then it will be entered as a new entry to the Plan Library.

5.3.2 Best-Plan Selection

The Best-Plan Selection builds on top of the greedy-plan selection by collecting all the plans from the library that are applicable to the task, and selecting the one with the best *success rate*.

This approach is designed such that it could be extended, by introducing exploration such that each plan is tested for at least a minumum amount before selecting them based on success rate. This, together with the *replaceability degree* could lead to removal of the plan from the library, helping to minimise the time spent searching for plans in the library. There would be many other approaches that could be integrated from the field of reinforcement learning and probabilistic planning that would cater to this approach, but due to COVID19 restrictions, we were not able to evaluate in a real world environment.

5.4 Empirical Evaluation

In order to test the effects of the Plan Library, we have used ROSPlan's Simulated Actions feature. Our original evaluation was to be completed with a robot, but this was not possible due to Covid-19 restrictions at the time of our evaluation. This feature allows the simulation of actions being dispatched and executed, letting the user specify a probability of action failure, and it implies that we are executing the same code we would use on a real robot, meaning that this can easily be transferred to real-world robot experiments, rather than simulated. However, due to working with simulated actions there were no behaviors that could be learned during executions, and as such all the results are based on *greedy-plan selection*.

In a real environment, actions would have different chances of failures, and would have led to the Plan Library to develop in a more targeted direction, and for replanning to be able to find solutions that avoid issues that happened in repeatead manners, as we could have evaluated more plan selection techniques. This would not be able with the current implementation of simulated actions, as we would not be able to replicate an issue or failures with an action that only happened when several factors were true at the same time, as they would appear in real life.

Algorithm 6 Best-Plan Selection

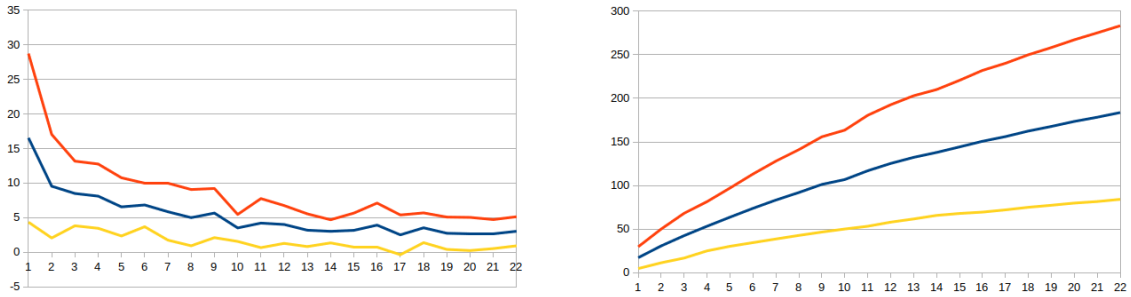
Input: A problem instance $\mathscr{P}$, a dictionary of previously solved planning tasks `PlanLib`
Output: A plan π

```
1: (init_state, goal) ← 𝒫;
2: plan_found ← False;
3: plan_pool ← [];
4: π ← empty;
5: for entry in PlanLib do
6:     if entry[init_state] matches init_state, and entry[goal] in goal then
7:         plan_pool.append(entry);
8:         plan_found ← True;
9:     end if
10: end for
11: if plan_found then
12:     best_rate ← 0
13:     for entry in plan_pool do
14:         if entry[success_rate] > best_rate then
15:             π ← entry[π];
16:         end if
17:     end for
18: else
19:     π ← CallPlannerInterface(𝒫);
20:     PlanLib.append(init_state, goal, π)
21: end if
22: return π
```

5.4.1 Experiment Design

For evaluation, we used a temporal implementation of the Office domain that we described as our running example in the second chapter. It consists of a robot-assistant that helps in a dynamic office setting (depicted in Figure 2.10).

The robot is tasked with navigating the environment and bringing different office resources (e.g.: mugs, post or papers) to the people in it, asking humans for help when needed. We created 10 problems, each having solutions that would need more steps to complete the task than the one prior which was done by increasing the number of predicates needed in the goal condition and increasing the number of items in the environment. They would end up taking between 3 to 20 seconds to compute, and would have a length that varied from 40 to 140 actions.



(a) (b)

Fig. 5.2 Summary plots averaged across all action failure probabilities on (a) Total planning time in seconds; (b) Number of plan rules in the plan library. Average values in blue, with +1 standard deviation (red) and -1 standard deviation (yellow). The averages come from the 5 problems which achieved this, all completing at least 22 runs. The rest did not have at least 10 runs completed across all action failure probabilities.

Each of these problems was then run with varying probabilities of each individual action failing. This probability was varied between 0.5 and 0.9. Higher probabilities of action failure meant, naturally, that more replanning was required. We ran each problem 40 times sequentially, meaning that the plan library was not cleared between these iterations, allowing the robot to learn through additions to the plan library[2]. We compare our method with a standard version of ROSPlan without a Plan Library.

We used POPF [19] with a timeout of 30 seconds to compute the plans. The overall system was given 500 seconds for each problem to be solved and reach the goal, timing out when this limit was reached. Tests were performed on a cluster utilising 26 Intel Xeon E5-2660 V4 (2.00GHz) processors and 192 GB of RAM.

All the experiments were executed inside Docker containers, so that different instances would not interfere with each other when running them in parallel.

5.4.2 Results

Over the course of our experiments, the modified version of ROSPlan with the plan library reached the goal 1403 times out of the 2000 problem instances it faced (timing out the remaining times). The system spent an average of 0.022 seconds (7.2% of the total planning time)

[2]Each problem/probability pair started from an empty plan library, so there was no learning between different experimental conditions

searching for plans in the Plan Library. The problem requested from the system was found in the Plan Library 64.6% of the times in total.

In comparison, the standard version of ROSPlan managed to reach the goal 1645 times, managing to perform better than the Plan Library version in the problems with more actions (100+). This is due to the fact that the Plan Library was being used blindly, exploiting only the plans it had solved already, with no exploration, meaning that if it got a poor plan in a prior run, it had a higher chance of getting stuck in it. This, together with higher action probability, would lead the agent into states that would need more than the allocated planning time (30 seconds) to solve, causing it to fail to reach the goal.

From Figure 5.2, we can see how the plan library performed overall. In short, across all the problems, and all the probabilities of action failure, the plan library works effectively.

Figure 5.2a shows that over successive runs, the cost of planning falls, while Figure 5.2b shows that the plan library grows, but showing signs that growth will flatten out. The second of these is exactly what we would expect, and the first is exactly what we would hope.

Similar to the Korf Conjuncture [84], where he states that the more memory is used for creating an abstraction-based heuristic leads to shorter time searching for a solution to the task, we also observe that the larger a plan library's, and to an extent the more memory used for it, the less time will be spent generating plans.

5.4.3 Individual Experiment Results

In this part, we will offer a detailed report and examinations of the results received from the experiments we carried out.

As discussed in Section 5.4.2, we ran each of our problems with probabilities of action failure that were set to 0.5, 0.6, 0.7, 0.8, and 0.9. The figures below show the total time used in planning for each problem and each probability of action failure, grouped by probability. That is, Figure 5.3 shows results for all the problems when the probability of action failure is 0.5, Figure 5.4 shows the results when the probability of action failure is 0.6, and Figures 5.5, 5.6 and 5.7 show the results for probabilities 0.7, 0.8, and 0.9 respectively.

Each graph plots two things. First, the time taken for planning when the plan library was (dark blue line) and was not (red line) used. Time is plotted on the left-hand y-axis. It is clear that using the plan library consistently reduces the time spent planning. That is, it reduces the

run time of the whole plan library node, including both using the plan library and running the planner (if required). The second thing that each graph plots is the proportion of plans that came from the plan library (dotted light blue line). The proportion is plotted on the right-hand y-axis.

We also provide summary plots that average results over each problem. That is, there is one plot per problem, where values are averaged over all probabilities. There are two plots, Figure 5.8 which reports the average time spent planning, and Figure 5.9, which reports the average size of the plan library. For both measures, we only provide graphs for problems where the plan library version of ROSPlan completed at least 5 runs since we do not think that averages over fewer runs provide useful information.

We also provide Table 5.1, which gives the correlation between the time spent planning and the number of replans for different action failure probabilities. See that there is a high positive correlation between total planning time and the number of replans for most cases where the Plan Library is not used (std), and a high negative correlation for most of the cases where the Plan Library is used (w/PL). In other words, the time spent planning tends to increase when replanning without the Plan Library, and tends to go down with the Plan Library.

ROSPlan	Action probability	1	2	3	4	5	6	7	8	9	10
std	0.5	0.9129	0.8196	0.8674	0.8674	0.7863	0.6867	0.0987	0.0542	0.5668	NA
	0.6	0.875	0.8651	0.8878	0.8878	0.8629	0.617	0.6031	0.7296	0.7296	0.5574
	0.7	0.83	0.8707	0.866	0.866	0.7885	0.7175	0.5856	0.7972	0.832	0.7703
	0.8	0.8889	0.8431	0.822	0.822	0.7845	0.7679	0.7789	0.8445	0.7096	0.8337
	0.9	0.9215	0.8485	0.7034	0.7034	0.8441	0.6897	0.4604	0.83811	0.8092	0.7806
w/ PlanLib	0.5	-0.8958	-0.8469	-0.9304	-0.2707	-0.625	-0.6968	-0.626	-0.7349	-0.996	0.7418
	0.6	-0.7796	-0.8154	-0.8189	-0.6881	-0.7508	-0.8085	-0.8833	-0.8021	-0.6647	-0.5173
	0.7	-0.8865	-0.8558	-0.8022	-0.7202	-0.7464	-0.8071	-0.6029	-0.79	-0.4865	-0.4274
	0.8	-0.831	-0.9163	-0.7604	-0.8468	-0.7542	-0.7294	-0.5599	-0.7376	-0.6412	-0.6416
	0.9	-0.7921	-0.865	-0.6316	-0.6623	-0.639	-0.6645	-0.7247	-0.6952	-0.564	-0.4067

Table 5.1 The correlation between the total time that the robot spent planning, and the numbers of re-plans. The correlation is shown per problem and per probability of failure, both with (w/PL) and without (std) the plan library.

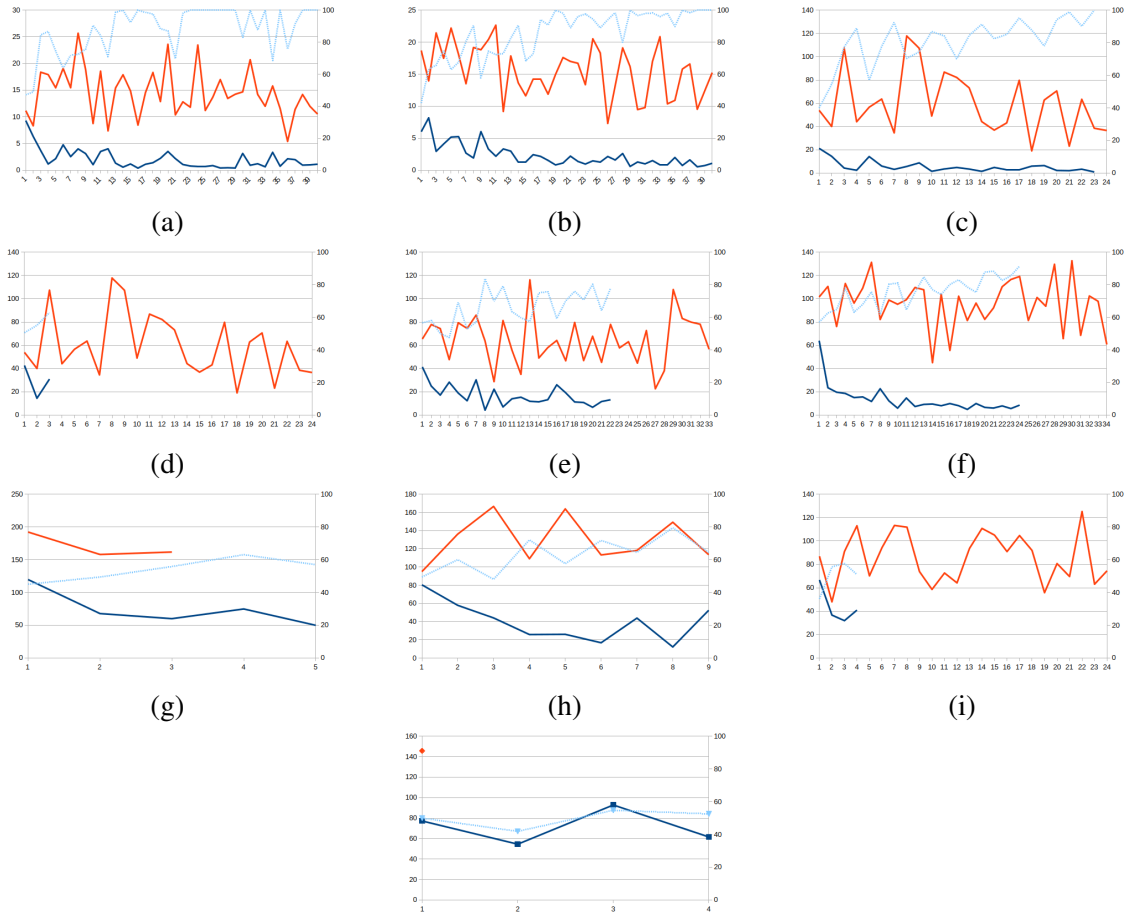
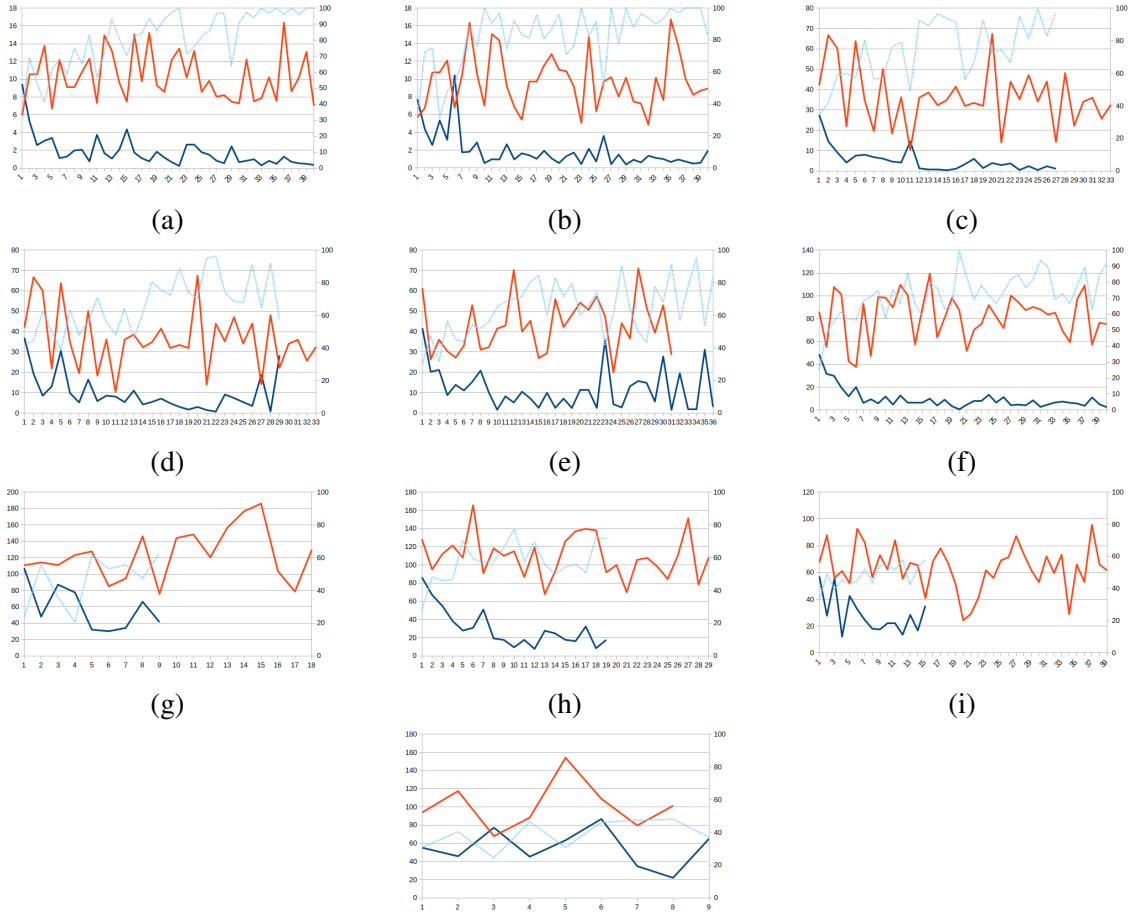


Fig. 5.3 Each graph represents a problem that we ran our experiments on, with action probability 0.5. The dark blue and red lines represent the total planning time for the solution running with and without the Plan Library node (the time is represented in seconds, and can be seen on the first y-axis). The dotted blue line represents how much the plan library was used from the total replans (the percentage is on the second y-axis). (a) Problem 1, (b) Problem 2, (c) Problem 3, (d) Problem 4, (e) Problem 5, (f) Problem 6, (g) Problem 7, (h) Problem 8, (i) Problem 9 and (j) Problem 10.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j)

Fig. 5.4 Each graph represents a problem that we ran our experiments on, with action probability 0.6. The dark blue and red lines represent the total planning time for the solution running with and without the Plan Library node (the time is represented in seconds, and can be seen on the first y-axis). The dotted blue line represents how much the plan library was used from the total replans (the percentage is on the second y-axis). (a) Problem 1, (b) Problem 2, (c) Problem 3, (d) Problem 4, (e) Problem 5, (f) Problem 6, (g) Problem 7, (h) Problem 8, (i) Problem 9 and (j) Problem 10.

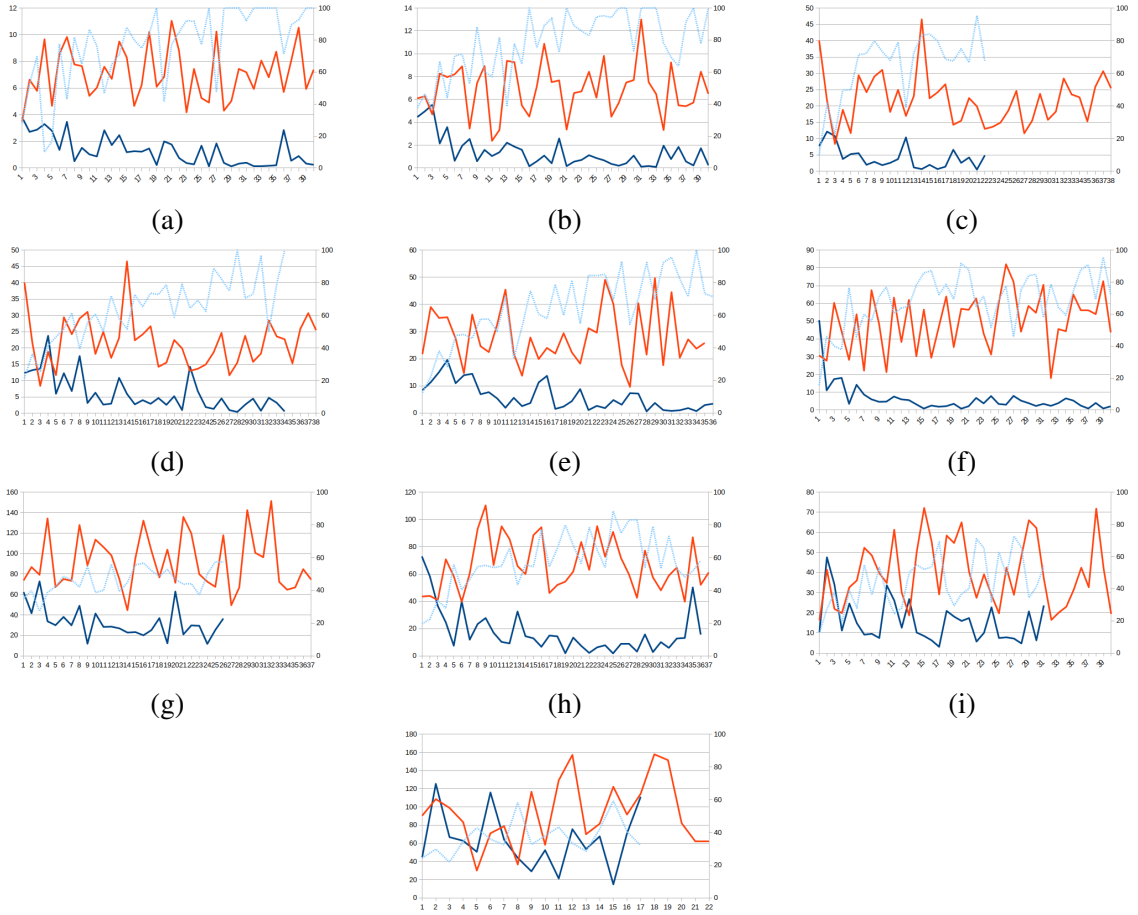
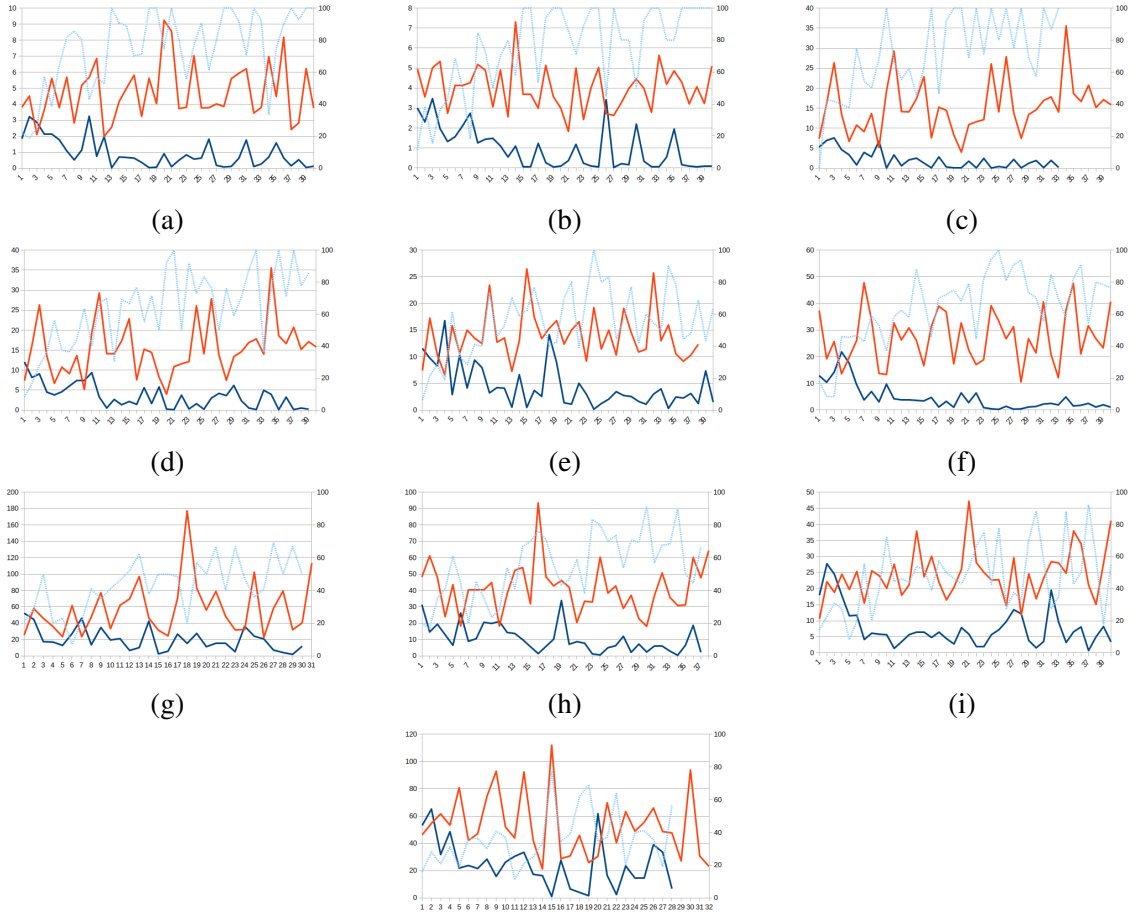


Fig. 5.5 Each graph represents a problem that we ran our experiments on, with action probability 0.7. The dark blue and red lines represent the total planning time for the solution running with and without the Plan Library node (the time is represented in seconds, and can be seen on the first y-axis). The dotted blue line represents how much the plan library was used from the total replans (the percentage is on the second y-axis). (a) Problem 1, (b) Problem 2, (c) Problem 3, (d) Problem 4, (e) Problem 5, (f) Problem 6, (g) Problem 7, (h) Problem 8, (i) Problem 9 and (j) Problem 10.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j)

Fig. 5.6 Each graph represents a problem that we ran our experiments on, with action probability 0.8. The dark blue and red lines represent the total planning time for the solution running with and without the Plan Library node (the time is represented in seconds, and can be seen on the first y-axis). The dotted blue line represents how much the plan library was used from the total replans (the percentage is on the second y-axis). (a) Problem 1, (b) Problem 2, (c) Problem 3, (d) Problem 4, (e) Problem 5, (f) Problem 6, (g) Problem 7, (h) Problem 8, (i) Problem 9 and (j) Problem 10.

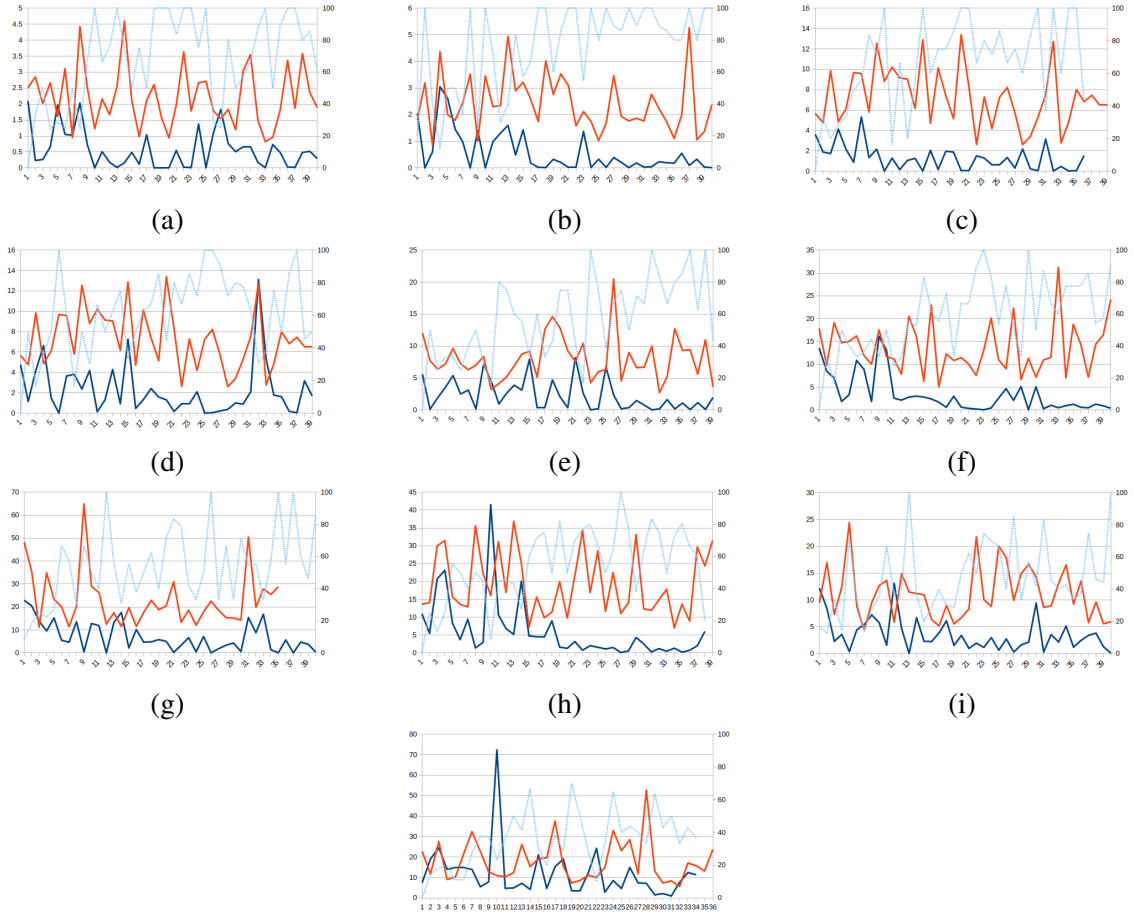
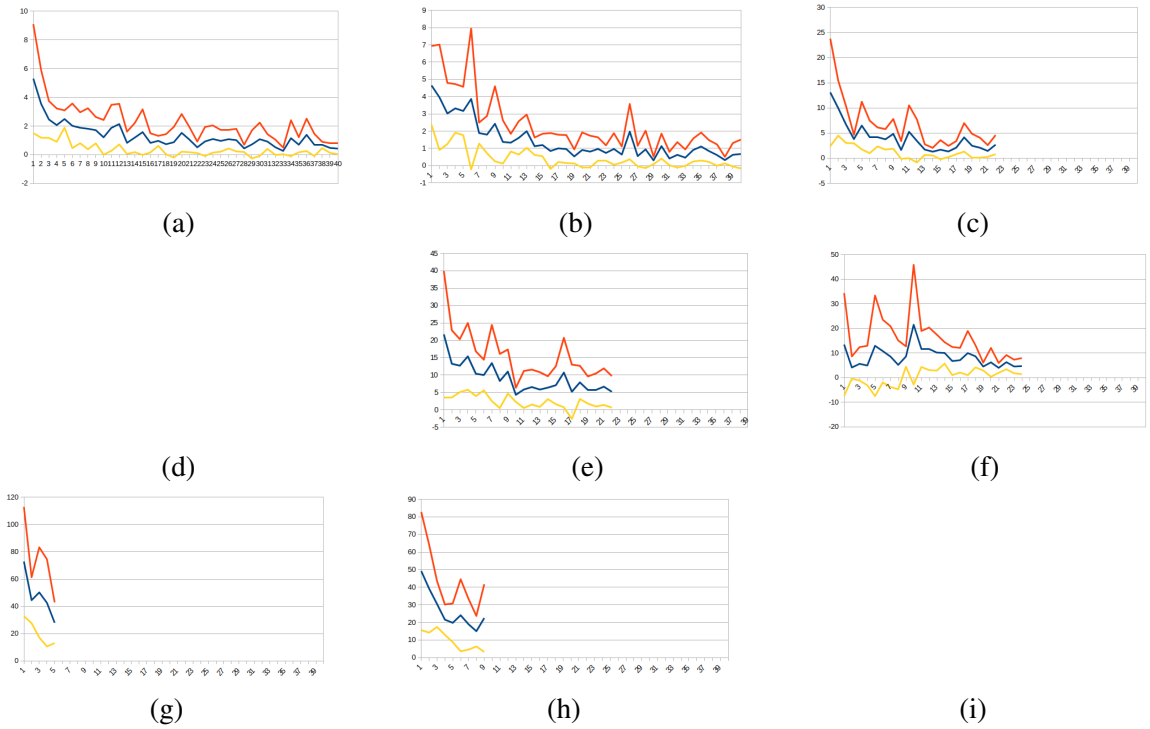


Fig. 5.7 Each graph represents a problem that we ran our experiments on, with action probability 0.9. The dark blue and red lines represent the total planning time for the solution running with and without the Plan Library node (the time is represented in seconds, and can be seen on the first y-axis). The dotted blue line represents how much the plan library was used from the total replans (the percentage is on the second y-axis). (a) Problem 1, (b) Problem 2, (c) Problem 3, (d) Problem 4, (e) Problem 5, (f) Problem 6, (g) Problem 7, (h) Problem 8, (i) Problem 9 and (j) Problem 10.



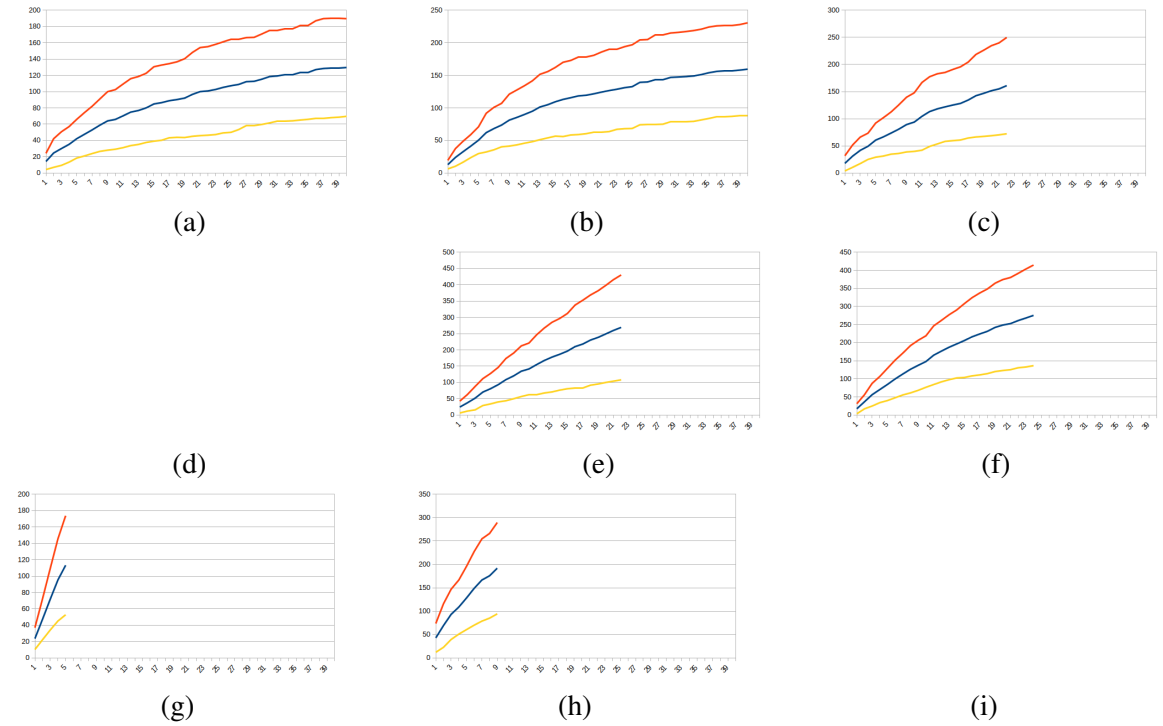
(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j)

Fig. 5.8 Summary plots of time spent planning in each of the problems, averaged across the probabilities of action failures 0.5, 0.6. 0.7, 0.8 and 0.9. Each graph shows the average (blue) as well as +1 (red) and -1 (yellow) standard deviation. (a) Problem 1, (b) Problem 2, (c) Problem 3, (d) Problem 4, (e) Problem 5, (f) Problem 6, (g) Problem 7, (h) Problem 8, (i) Problem 9 and (j) Problem 10. A missing graph indicates that there were less than 5 completed runs, making an average misleading — we keep the blanks to make easy comparisons across the figures.



(a) (b) (c) (d) (e) (f) (g) (h) (i) (j)

Fig. 5.9 Summary plots of plan library size in each of the problems, averaged across the probabilities of action failures 0.5, 0.6. 0.7, 0.8 and 0.9. Each graph shows the average (blue) as well as +1 (red) and -1 (yellow) standard deviation. (a) Problem 1, (b) Problem 2, (c) Problem 3, (d) Problem 4, (e) Problem 5, (f) Problem 6, (g) Problem 7, (h) Problem 8, (i) Problem 9 and (j) Problem 10. A missing graph indicates that there were less than 5 completed runs, making an average misleading — we keep the blanks to make easy comparisons across the figures.

5.5 Conclusion

In this chapter we achieved our final objective – **O3** – and the final two contributes – **C5** and **C6**. Results seen across our experiments reinforce our conclusions about the effectiveness of the plan library. The results for individual experiments show that, consistently across the problems and probabilities of action failure: 1) using the Plan Library reduces the time spent

planning, and 2) the use of the Plan Library rises over time but shows signs of converging into a maximum needed size once it has executed enough plans in the environment.

Overall, we show that this is an effective mechanism for introduction of recycled plans in general execution and saves time compared to relying solely on replanning. We hypothesized that once tested in a real environment, this approach would be able to avoid specific issues in the environment, without the need of changing the description of the environment.

When we think of the deliberation that we ourselves do, we rarely solve a task from scratch, instead we combine planning from first principles with experience. We propose a solution that is a step towards mirroring this approach to problem-solving, reusing past plans where possible, and replanning from scratch where it is not.

Our results and contributions – **C5** and **C6** – show that for a dynamic environment and medium length tasks, this is an approach that, after a short number of runs, manages to gain enough experience for a considerable speed up in deliberation to emerge.

In the next chapter, we will go over the conclusion that can be taken from this thesis and seeing how our objectives have been met. We will also set up the future works that are created by our work, and how they can be combined to find a better reasoning system for robotic agents.

Chapter 6

Conclusion and Future Work

The focus of this thesis was to investigate and find new ways of using memory-based systems to improve plan generation and execution. We aimed to study the current state-of-the-art planners that solved classical planning problems optimally, and show what can be learned from this area and extended to agents reasoning in dynamic environments.

In this chapter, we will offer a short summary of what we presented across this thesis, and look forward to how this body of work can be continued.

6.1 Objectives Evaluation

O1: In Chapter 3, we focused on solving the first objective of this thesis, by investigating the *Pattern Selection* problem – **C2**. It entails combining several Pattern Databases into one best heuristic for solving a planning task optimally. We built on top of the Complementary Pattern Collection heuristic presented by Franco et al. [43], and modified their bin-packing subroutine by using greedy algorithms such as First Fit Increasing/Decreasing or by using Constraint Programming solvers and modelling languages such as MiniZinc for find solutions to the problem.

Another addition was the Partial-Gamer approach for building one best pattern for solving a task. This, combined with a greedy bin-packing solution has shown to be offering the best results across all existing planning domains. The resulting planners showed state-of-the-art

performance across all available planning benchmarks, performing especially well on the most modern sets from IPC11, IPC14 and IPC18 – **C1**.

O2: Chapter 4 set out to do an analysis of the Cost-Optimal track from the 2018 Deterministic International Planning Competition, where we had two of our planners inside the top four and inside 2% from the eventual competition winner. This competition reinforced the conclusions from the previous edition, that uninformed symbolic search is a great solution to domain-independent planning, but there are still domains that it cannot perform, failing to offer a good domain coverage on them.

We did see that the combination of symbolic search and Pattern Databases do cover the gaps in coverage left by uninformed symbolic search, with all single planner solutions in the top five relying on PDBs while the top three rely on PDBs and symbolic search.

The surprising conclusion from the competition was that portofolio-planning, for which we offered a definition in the chapter, has emerged as a valid solution to cost-optimal planning. The winner of the competition showed that planning problems can be parsed into different features that can help train classifiers for mapping tasks to the planners that have the best chance of solving them. Portofolio-planning is a contentious subject for future competitions, but it does show that it is an area that should be investigated more, with different tracks for future competitions – **C3**.

Also in this chapter we provided a new metric, *normalised coverage* for measuring the domain-independence of a planner. This was due to the varying number of problems per planning domain available, biasing the conclusions that are taken by looking only at the total number of problems solved –**C4**.

O3: Finally, in Chapter 5 we focused on the field of planning under uncertainty for robotic agents, and investigated how Plan Libraries, an idea inspired by Belief-Desire-Intention agents, can be used to make deterministic planning a fast and robust solution. We proposed a new ROS node, Plan Library, to be added to the ROSPlan framework, and showed how it can affect reasoning during long-term deployments – **C5**. We also proposed two different methods of selecting plans from a set that could be executed for a planning task, Greedy-Plan Selection and Best-Plan Selection – **C6**. Our results reinforce the need for model-based reasoning systems to fully utilise memory across executions, as more memory used leads to less time reasoning.

6.2 Future Work

6.2.1 Pattern Database

The first idea that would combine both main ideas from this thesis would be the generation of reusable Pattern Databases. This could be done with an analysis of goals, generating many small PDBs and then selecting appropriate ones depending on which task is submitted to the planner.

Another work that we are interested in looking into Saturated Cost-Partitioning and other methods that could increase the heuristic estimates across a PDB. This, together with a study into which cost-partitioning method works best on what type of planning task, could lead to developing cost-partitioning portofolios inside a planner using PDBs.

6.2.2 Plan Library

Our experiments made the big assumption that all actions fail with the same probability. We aim to run our experiments in real world environment, where action failure is a property of the world instead of being it defined in our simulation. This will give us a better idea of how often a Plan Library can be used, and how fast it can accumulate knowledge about the environment. Seeing if we can balance exploitation of past plans with exploration to discover new plans would be complementary to this work.

Knowing that the time spent searching for a plan is short (7.2% of all planning time), we would like to investigate if it is possible to add planners that search for better quality plans. Comparisons between different types of heuristics will tell us if using a Plan Library makes it possible to use classical optimal planners in planning for robotics.

Any future work should investigate more agent exploration methods, for generating a robust Plan Library. Methods from reinforcement learning should be explored and compared for this.

Another approach would be to implement a system similar to the ones in portofolio-planners and map different tasks to different planners. This approach could be PDDL agnostic, and lead to having portofolios that include planners using different input languages. Some tasks could benefit from reasoning about the temporal actions, while others would not, so solving it with a classical planner would work better.

Finally, we are investigating if having a library of the robot's abilities would increase the explainability of its reasoning process. Given such a library, the robot would be able to keep track of its executions, giving a more in-depth explanation for its decision based on its *experiences*.

References

[1] Ambros-Ingerson, J. A. and Steel, S. (1988). Integrating planning, execution and monitoring. In *AAAI*, volume 88, pages 21–26.

[2] Bacchus, F. and Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artif. Intell.*, 116(1-2):123–191.

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

[4] Barley, M., Franco, S., and Riddle, P. (2014). Overcoming the utility problem in heuristic generation: Why time matters. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*.

[5] Bonet, B. (2013). An admissible heuristic for sas+ planning obtained from the state equation. In *IJCAI*, pages 2268–2274.

[6] Bonet, B. and Geffner, H. (1999). Planning as heuristic search: New results. In *European Conference on Planning*, pages 360–372. Springer.

[7] Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons.

[8] Bratman, M. et al. (1987). *Intention, plans, and practical reason*, volume 10. Harvard University Press Cambridge, MA.

[9] Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691.

[10] Canal, G., Cashmore, M., Krivić, S., Alenyà, G., Magazzeni, D., and Torras, C. (2019). Probabilistic planning for robotics with rosplan. In *Annual Conference Towards Autonomous Robotic Systems*, pages 236–250. Springer.

[11] Cashmore, M., Fox, M., Larkworthy, T., Long, D., and Magazzeni, D. (2014). Auv mission control via temporal planning. In *2014 IEEE international conference on robotics and automation (ICRA)*, pages 6535–6541. IEEE.

[12] Cashmore, M., Fox, M., Long, D., and Magazzeni, D. (2016). A compilation of the full PDDL+ language into smt. In *AAAI Workshop: Planning for Hybrid Systems*.

[13] Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., Carrera, A., Palomeras, N., Hurtos, N., and Carreras, M. (2015). ROSPlan: Planning in the robot operating system. In *ICAPS*, pages 333–341.

[14] Cashmore, M., Magazzeni, D., and Zehtabi, P. (2020). Planning for hybrid systems via satisfiability modulo theories. *Journal of Artificial Intelligence Research*, 67:235–283.

[15] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169.

[16] Clarke, E. M., McMillan, K. L., Campos, S. V. A., and Hartonas-Garmhausen, V. (1996). Symbolic model checking. In *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, pages 419–427.

[17] Coles, A., Coles, A., Olaya, A. G., Jiménez, S., López, C. L., Sanner, S., and Yoon, S. (2012). A survey of the seventh international planning competition. *Ai Magazine*, 33(1):83–88.

[18] Coles, A., Fox, M., Long, D., and Smith, A. (2008). Additive-disjunctive heuristics for optimal planning. In *ICAPS*, pages 44–51.

[19] Coles, A. J., Coles, A. I., Fox, M., and Long, D. (2010). Forward-chaining partial-order planning. In *ICAPS*.

[20] Culberson, J. and Schaeffer, J. (1994). Efficiently searching the 15-puzzle.

[21] Culberson, J. C. and Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, 14(4):318–334.

[22] Derval, G., Regin, J.-C., and Schaus, P. (2017). Improved filtering for the bin-packing with cardinality constraint. In *CP*.

[23] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

[24] Dósa, G. (2007). The tight bound of first fit decreasing bin-packing algorithm is ffd (i)<= 11/9opt (i)+ 6/9. In *International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 1–11. Springer.

[25] Edelkamp, S. (2001). Planning with pattern databases. In *European Conference on Planning (ECP)*.

[26] Edelkamp, S. (2002a). Symbolic pattern databases in heuristic search planning. pages 274–293.

[27] Edelkamp, S. (2002b). Symbolic pattern databases in heuristic search planning. In *AIPS*, pages 274–283.

[28] Edelkamp, S. (2006). Automated creation of pattern database search heuristics. In *International Workshop on Model Checking and Artificial Intelligence*, pages 35–50. Springer.

[29] Edelkamp, S. (2014). Planning with pattern databases. In *Sixth European Conference on Planning*.

[30] Edelkamp, S. and Hoffmann, J. (2004). Pddl2. 2: The language for the classical part of the 4th international planning competition. Technical report, Technical Report 195, University of Freiburg.

[31] Edelkamp, S. and Kissmann, P. (2011). On the complexity of BDDs for state space search: A case study in Connect Four. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*.

[32] Edelkamp, S., Kissmann, P., and Torralba, Á. (2015). BDDs strike back (in AI planning). In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 4320–4321.

[33] Edelkamp, S. and Moraru, I. (2019). Cost-optimal planning in the ipc 2018: Symbolic search and planning pattern databases vs. portfolio planning. *WIPC 2019*, page 15.

[34] Edelkamp, S. and Reffel, F. (1998). OBDDs in heuristic search. In *KI-98: Advances in Artificial Intelligence, 22nd Annual German Conference on Artificial Intelligence, Bremen, Germany, September 15-17, 1998, Proceedings*, pages 81–92.

[35] Edelkamp, S. and Schroedl, S. (2011). *Heuristic search: theory and applications*. Elsevier.

[36] Estlin, T., Gaines, D., Chouinard, C., Castano, R., Bornstein, B., Judd, M., Nesnas, I., and Anderson, R. (2007). Increased mars rover autonomy using ai planning, scheduling and execution. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 4911–4918. IEEE.

[37] Felner, A., Korf, R. E., and Hanan, S. (2004). Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318.

[38] Felner, A. and Ofek, N. (2007). Combining perimeter search and pattern database abstractions. In *Abstraction, Reformulation, and Approximation, 7th International Symposium, SARA 2007, Whistler, Canada, July 18-21, 2007, Proceedings*, pages 155–168.

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

[40] Fox, M. and Long, D. (2002). Pddl+: Modeling continuous time dependent effects. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, volume 4, page 34.

[41] Fox, M. and Long, D. (2003). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR*, 20:61–124.

[42] Franco, S., Lelis, L. H., Barley, M., Edelkamp, S., Martines, M., and Moraru, I. (2018). The complementary2 planner in the ipc 2018. *IPC-9 planner abstracts*, pages 28–31.

[43] Franco, S., Torralba, Á., Lelis, L. H. S., and Barley, M. (2017). On creating complementary pattern databases. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 4302–4309.

[44] Frohm, J., Lindström, V., Winroth, M., and Stahre, J. (2008). Levels of automation in manufacturing. *Ergonomia*.

[45] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractibility, A Guide to the Theory of NP-Completeness*. Freeman & Company.

[46] Gaschnig, J. (1979). A problem similarity approach to devising heuristics: First results. In *ijcai*, pages 434–441.

[47] Gat, E., Bonnasso, R. P., Murphy, R., et al. (1998). On three-layer architectures. *Artificial intelligence and mobile robots*, 195:210.

[48] Gecode Team (2006). Gecode: Generic constraint development environment. Available from `http://www.gecode.org`.

[49] Geffner, H. (2014). Artificial intelligence: From programs to solvers. *AI Communications*, 27(1):45–51.

[50] Geffner, P. H. H. and Haslum, P. (2000). Admissible heuristics for optimal planning. In *Proceedings of the 5th Internat. Conf. of AI Planning Systems (AIPS 2000)*, pages 140–149.

[51] Gerevini, A. and Long, D. (2005). Plan constraints and preferences in pddl3. Technical report, Technical Report 2005-08-07, Department of Electronics for Automation ....

[52] Gurobi Optimization, I. (2016). Gurobi optimizer reference manual.

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

[54] Haslum, P., Bonet, B., Geffner, H., et al. (2005). New admissible heuristics for domain-independent planning. In *AAAI*, volume 5, pages 9–13.

[55] Haslum, P., Botea, A., Helmert, M., Bonet, B., and Koenig, S. (2007a). Domain-independent construction of pattern database heuristics for cost-optimal planning. pages 1007–1012.

[56] Haslum, P., Botea, A., Helmert, M., Bonet, B., Koenig, S., et al. (2007b). Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*, volume 7, pages 1007–1012.

[57] Haslum, P., Lipovetzky, N., Magazzeni, D., and Muise, C. (2019). An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2):1–187.

[58] Haugeland, J. (1989). *Artificial intelligence: The very idea*. MIT press.

[59] Hayes, P. J. (1975). A representation for robot plans. In *Proceedings of the 4th international joint conference on Artificial intelligence-Volume 1*, pages 181–188.

[60] Helmert, M. (2004). A planning heuristic based on causal graph analysis. pages 161–170.

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

[62] Helmert, M. (2008). *Understanding Planning Tasks: Domain Complexity and Heuristic Decomposition*, volume 4929 of *Lecture Notes in Computer Science*. Springer.

[63] Helmert, M. and Domshlak, C. (2009). Landmarks, critical paths and abstractions: what's the difference anyway? In *ICAPS*, pages 162–169.

[64] Helmert, M., Haslum, P., Hoffmann, J., et al. (2007). Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, pages 176–183.

[65] Helmert, M. and Lasinger, H. (2010). The scanalyzer domain: Greenhouse logistics as a planning problem. In *Twentieth International Conference on Automated Planning and Scheduling*.

[66] Helmert, M., Röger, G., and Karpas, E. (2011). Fast downward stone soup: A baseline for building planner portfolios. In *ICAPS 2011 Workshop on Planning and Learning*, pages 28–35.

[67] Hendler, J. A., Tate, A., and Drummond, M. (1990). Ai planning: Systems and techniques. *AI Magazine*, 11(2):61.

[68] Hodges, W. (2010). Ibn sīnā on analysis: 1. proof search. or: abstract state machines as a tool for history of logic. In *Fields of Logic and Computation*, pages 354–404. Springer.

[69] Hoffmann, J. and Nebel, B. (2001). The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302.

[70] Holland, J. (1975). *Adaption in Natural and Artificial Systems*. PhD thesis, University of Michigan.

[71] Holte, R., Newton, J., Felner, A., Meshulam, R., and Furcy, D. (2004). Multiple pattern databases. pages 122–131.

[72] Holte, R. C. and Hernádvölgyi, I. T. (1999). A space-time tradeoff for memory-based heuristics. In *AAAI/IAAI*, pages 704–709. Citeseer.

[73] Holte, R. C. and Hernádvölgyi, I. T. (2001). A space-time tradeoff for memory-based heuristics.

[74] Howey, R., Long, D., and Fox, M. (2004). Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In *16th IEEE International Conference on Tools with Artificial Intelligence*, pages 294–301. IEEE.

[75] Huang, Z., Sklar, E., and Parsons, S. (2020). Design of automatic strawberry harvest robot suitable in complex environments. In *Companion of the 2020 ACM/IEEE International Conference on Human-Robot Interaction*, pages 567–569.

[76] Ingrand, F. and Ghallab, M. (2017). Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44.

[77] Ingrand, F. F., Georgeff, M. P., and Rao, A. S. (1992). An architecture for real-time reasoning and system control. *IEEE expert*, 7(6):34–44.

[78] Karpas, E. and Domshlak, C. (2009). Cost-optimal planning with landmarks. In *IJCAI*, pages 1728–1733.

[79] Karpas, E., Katz, M., and Markovitch, S. (2011). When optimal is just not good enough: Learning fast informative action cost partitionings. In *ICAPS*.

[80] Katz, M. and Domshlak, C. (2008). Optimal additive composition of abstraction-based admissible heuristics. In *ICAPS*, pages 174–181.

[81] Kautz, H. and Selman, B. (1996). Pushing the envelope: Planning propositional logic, and stochastic search. pages 1194–1201.

[82] Kissmann, P. and Edelkamp, S. (2011). Improving cost-optimal domain-independent symbolic planning. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*.

[83] Klößner, T., Hoffmann, J., Steinmetz, M., and Torralba, A. (2021). Pattern databases for goal-probability maximization in probabilistic planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 201–209.

[84] Korf, R. E. (1997). Finding optimal solutions to Rubik's Cube using pattern databases. pages 700–705.

[85] Korf, R. E. (2002). A new algorithm for optimal bin packing. pages 731–736.

[86] Korf, R. E. (2003). An improved algorithm for optimal bin packing. In *IJCAI*, pages 1252–1258.

[87] Korf, R. E. and Felner, A. (2002). *Chips Challenging Champions: Games, Computers and Artificial Intelligence*, chapter Disjoint Pattern Database Heuristics, pages 13–26. Elsevier.

[88] Kortenkamp, D., Schreckenghost, D., and Bonasso, R. (1998). Three nasa application domains for integrated planning, scheduling and execution.

[89] Krivic, S., Cashmore, M., Magazzeni, D., Szedmak, S., and Piater, J. (2020). Using machine learning for decreasing state uncertainty in planning. *Journal of Artificial Intelligence Research*, 69:765–806.

[90] Kvarnström, J. and Magnusson, M. (2003). TALplanner in the third international planning competition: Extensions and control rules. *J. Artif. Intell. Res.*, 20:343–377.

[91] Levi, L. H., Franco, S., Abisrror, M., Barley, M., Zilles, S., and Holte, R. (2016). Heuristic subset selection in classical planning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16)*, pages 3185–3195. AAAI Press/IJCAI.

[92] Mariott, K. and Stuckey, P. (1998). *Programming with Constraints*. MIT Press.

[93] Martinez, M., Moraru, I., Edelkamp, S., and Franco, S. (2018). Planning-PDBs planner in the IPC 2018. *IPC-9 Planner Abstracts*, pages 63–66.

[94] McCarthy, J. and Hayes, P. J. (1981). Some philosophical problems from the standpoint of artificial intelligence. In *Readings in artificial intelligence*, pages 431–450. Elsevier.

[95] McDermott, D. (1998). The 1998 ai planning systems competition. In *AI Magazine*, pages 35–55.

[96] McRae, J. N., Gay, C. J., Nielsen, B. M., and Hunt, A. P. (2019). Using an unmanned aircraft system (drone) to conduct a complex high altitude search and rescue operation: a case study. *Wilderness & environmental medicine*, 30(3):287–290.

[97] Meneguzzi, F. and De Silva, L. (2015). Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. *The Knowledge Engineering Review*, 30(1):1–44.

[98] Meneguzzi, F. and Luck, M. (2008). Leveraging new plans in agentspeak (pl). In *International Workshop on Declarative Agent Languages and Technologies*, pages 111–127. Springer.

[99] Minsky, M. (1961). Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30.

[100] Moraru, I., Canal, G., and Parsons, S. (2021). Using plan libraries for improved plan execution. In *UKRAS21 Conference:"Robotics at home" Proceedings*.

[101] Moraru, I. and Edelkamp, S. (2019). Benchmarks old and new: How to compare domain independence for costoptimal classical planning. In *ICAPS 2019 Workshop on the International Planning Competition (WIPC)*, pages 36–39.

[102] Moraru, I., Edelkamp, S., Franco, S., and Martinez, M. (2019a). Simplifying automated pattern selection for planning with symbolic pattern databases. In *Künstliche Intelligenz*, pages 249–263. Springer.

[103] Moraru, I., Edelkamp, S., Martinez, M., and Franco, S. (2019b). Simplifying automated pattern selection for planning with symbolic pattern databases. *HSDIP 2019*, page 46.

[104] Munoz, M. M., Moraru, I., and Edelkamp, S. J. (2018). Automated pattern selection using minizinc. In *Int'l Conference on Principles and Practice of Constraint Programming: Workshop of Constraints and AI Planning*.

[105] Nebel, B. and Koehler, J. (1995). Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, 76(1-2):427–454.

[106] Nechushtai, E. and Lewis, S. C. (2019). What kind of news gatekeepers do we want machines to be? filter bubbles, fragmentation, and the normative dimensions of algorithmic recommendations. *Computers in Human Behavior*, 90:298–307.

[107] Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. (2007). Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer.

[108] Nilsson, N. J. (1984). Shakey the robot. Technical report, SRI INTERNATIONAL MENLO PARK CA.

[109] Onyedinma, C., Gavigan, P., and Esfandiari, B. (2020). Toward campus mail delivery using bdi. *Journal of Sensor and Actuator Networks*, 9(4):56.

[110] Parberry, I. (2015). Solving the (n^2 - 1)-puzzle with 8/3 n^3 expected moves. *Algorithms*, 8(3):459–465.

[111] Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

[112] Pednault, E. P. (1994). Adl and the state-transition model of action. *Journal of logic and computation*, 4(5):467–512.

[113] Piotrowski, W. M., Fox, M., Long, D., Magazzeni, D., and Mercorio, F. (2016). Heuristic planning for pddl+ domains. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*.

[114] Piotrowski, W. M., Fox, M., Long, D., Magazzeni, D., and Mercorio, F. (2017). Pddl+ planning with temporal pattern databases. In *Workshops at the Thirty-First AAAI Conference on Artificial Intelligence*.

[115] Pommerening, F., Helmert, M., and Bonet, B. (2017a). Higher-dimensional potential heuristics for optimal classical planning.

[116] Pommerening, F., Helmert, M., and Bonet, B. (2017b). Higher-dimensional potential heuristics for optimal classical planning.

[117] Pommerening, F., Helmert, M., Röger, G., and Seipp, J. (2015). From non-negative to general operator cost partitioning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

[118] Preditis, A. (1993). Machine discovery of admissible heuristics. *Machine Learning*, 12:117–142.

[119] Pylyshyn, Z. W. (1987). The robot's dilemma: The frame problem in artificial intelligence.

[120] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*. Kobe, Japan.

[121] Rao, A. S. and Georgeff, M. P. (1995). Bdi agents: from theory to practice. In *Proceedings of the First International Conference on Multiagent Systems.*, pages 312–319.

[122] Richter, S. and Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res.*, 39:127–177.

[123] Robertson, J. and Young, R. M. (2015). Automated gameplay generation from declarative world representations. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

[124] Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

[125] Sandewall, E. and Rönnquist, R. (1986). *A representation of action structures*. Universitetet i Linköping/Tekniska Högskolan i Linköping. Institutionen för ....

[126] Sanner, S. (2010). Relational dynamic influence diagram language (rddl): Language description. *Unpublished ms. Australian National University*, 32:27.

[127] Seipp, J. and Helmert, M. (2018a). Counterexample-guided cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, 62:535–577.

[128] Seipp, J. and Helmert, M. (2018b). Counterexample-guided cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, 62:535–577.

[129] Seipp, J. and Helmert, M. (2019). Subset-saturated cost partitioning for optimal classical planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

[130] Seipp, J., Keller, T., and Helmert, M. (2017). A comparison of cost partitioning algorithms for optimal classical planning. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*.

[131] Sievers, S., Katz, M., Sohrabi, S., Samulowitz, H., and Ferber, P. (2019). Deep learning for cost-optimal planning: Task-dependent planner selection. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*.

[132] Slaney, J. K. and Thiébaux, S. (2001). Blocks world revisited. *Artif. Intell.*, 125(1-2):119–153.

[133] Smith, S. J., Nau, D., and Throop, T. (1998). Computer bridge: A big win for ai planning. *AI magazine*, 19(2):93–93.

[134] Srivastava, S., Immerman, N., and Zilberstein, S. (2011). A new representation and associated algorithms for generalized planning. *Artificial Intelligence*, 175(2):615–647.

[135] Torralba, Á., Alcázar, V., Kissmann, P., and Edelkamp, S. (2017). Efficient symbolic search for cost-optimal planning. *Artif. Intell.*, 242:52–79.

[136] Torralba, A., Seipp, J., and Sievers, S. (2021). Automatic instance generation for classical planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 376–384.

[137] Vallati, M., Chrpa, L., Grześ, M., McCluskey, T. L., Roberts, M., Sanner, S., et al. (2015). The 2014 international planning competition: Progress and trends. *Ai Magazine*, 36(3):90–98.

[138] Valtorta, M. (1984). A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences*, 34:48–59.

[139] Van Beek, P. and Chen, X. (1999). Cplan: A constraint programming approach to planning. In *AAAI/IAAI*, pages 585–590.

[140] Van Den Briel, M., Benton, J., Kambhampati, S., and Vossen, T. (2007). An lp-based heuristic for optimal planning. In *International Conference on Principles and Practice of Constraint Programming*, pages 651–665. Springer.

[141] Vossen, T., Ball, M., Lotem, A., and Nau, D. (2000). Applying integer programming to ai planning. *The Knowledge Engineering Review*, 15(1):85–100.

[142] Wah, B. W. and Chen, Y. (2004). Subgoal partitioning and global search for solving temporal planning problems in mixed space. *International Journal on Artificial Intelligence Tools*, 13(4):767–790.

[143] Xu, M., Bauters, K., McAreavey, K., and Liu, W. (2018). A framework for plan library evolution in bdi agent systems. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 414–421. IEEE.

[144] Yaghoubi, S., Akbarzadeh, N. A., Bazargani, S. S., Bazargani, S. S., Bamizan, M., and Asl, M. I. (2013). Autonomous robots for agricultural tasks and farm assignment and future trends in agro robots. *International Journal of Mechanical and Mechatronics Engineering*, 13(3):1–6.

[145] Yang, F., Culberson, J., Holte, R., Zahavi, U., and Felner, A. (2008). A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research*, 32:631–662.

[146] Yoon, S. W., Fern, A., and Givan, R. (2007). Ff-replan: A baseline for probabilistic planning.

[147] Younes, H. L. and Littman, M. L. (2004). Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162*, 2:99.