TASK SCHEDULING AND MERGING IN SPACE AND TIME

by

LENKA MUDROVÁ

A thesis submitted to
University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
June 2017

ABSTRACT

Every day, robots are being deployed in more challenging environments, where they are required to perform complex tasks. In order to achieve these tasks, robots rely on intelligent deliberation algorithms. In this thesis, we study two deliberation approaches – task scheduling and task planning. We extend these approaches in order to not only deal with temporal and spatial constraints imposed by the environment, but also exploit them to be more efficient than the state-of-the-art approaches.

Our first main contribution is a scheduler that exploits a heuristic based on Allen's interval algebra to prune the search space to be traversed by a mixed integer program. We empirically show that the proposed scheduler outperforms the state of the art by at least one order of magnitude. Furthermore, the scheduler has been deployed on several mobile robots in long-term autonomy scenarios.

Our second main contribution is the POPMER$_X$ algorithm, which is based on merging of partially ordered temporal plans. POPMER$_X$ first reasons with the spatial and temporal structure of separately generated plans. Then, it merges these plans into a single final plan, while optimising the makespan of the merged plan. We empirically show that POPMER$_X$ produces better plans that the-state-of-the-art planners on temporal domains with time windows. Note that the results presented in this thesis have been significantly improved. Please refer to our journal paper.

To girls and women around the world,
who are told that they cannot follow their
dreams.

ACKNOWLEDGEMENT

My deepest gratitude goes to my supervisor Dr. Nick Hawes. He was the best supervisor I could hope for. I do not have enough space here to list all the things he did to support my research and personal growth. I would thus like to highlight the most important ones. First, I am grateful for the amount of freedom I got to direct the scope of this thesis. Second, he was always there for me to listen and to help when I was getting lost. He showed me that there is a life beyond research.

I am very grateful for all my colleagues and friends. A special thanks goes out to Dr. Bruno Lacerda, who was my second supervisor. He was always ready to give me advice. Also, he was my regular partner in crime during long nights before paper submission deadlines. He was also great to hangout and have fun with. My gratitude goes to Prof. Manuela Veloso from Carnegie Mellon University for hosting me during my research visit. The visit of her group was life-changing. I would like to also thank Dr. Lars Kunze and Dr. Chris Burbridge for advice especially at the beginning of my studies. Furthermore, I would like to thank Prof. Jeremy Wyatt and Dr. Mark Lee for valuable feedback on my progress, all members of the STRANDS project, who made this project a big success and my experience wholly enjoyable and all the participants of the Örebro Winter School and the IEEE-RAS Summer School for their ideas and the fun we had together. Last but not least, I would like to thank all members of the CORAL group at CMU who made my stay pleasant.

I would have never started this PhD if I had not met Jaroslav Halgašík. His support was critical for my decision to go study abroad. My life could have been also completely different if Mrs. Molitorisová had not been my math teacher in middle school. She supported me in my rebellion against social norms when I chose to study at a high school focused on electrical engineering. I am also very grateful to Dr. Jan Faigl, who was my supervisor for my bachelor and master thesis, for all his feedback regarding my academic work and for his support.

I would also like to thank those outside of academia. I would like to say thank you to my partner Christoph for what he is and for his continued support; to my parents for all their support throughout my life and for managing to stay sane when I moved out; to Rusty, our dog, for removing my stress; to my brother Olda, who has been my best friend since I was born; to his wife Radana for making him happy; and especially to my nephews Oldík and Filípek for their warm hugs and never ending supply of positive energy.

My life would not be same without my friends. A special thank you goes out to Lenka Caletková and Nerea Luis for not only being my soul mates, but also for all the great laughs and cries we shared. I am very grateful to all my friends from the University of Birmingham: to Mohab Elkaref, Bram Geron and Rusen Umit for being ready to listen to me any time; to Tomáš Jakl, Christian Reuther, Cory Knapp, Michael Denzel, Fani Tsapeli, Yuning Feng, Ferdian Jovan and Khulood AlYahya for all hang outs. Thank you to my friends outside of the Computer Science building: Giedrius Kudelis, Elisa Lauricell and Denise Hebersberger; and to my previous classmates: Martina Jirkalová, Honza Černý, Ondra Marcinka, Ondra Hrstka, Tomáš Juchelka, Honza Zábojník, Petr Vaněk, Aleš Ondráček, Filip Jareš, Tereza Hyková, Karol Bujaček, Honza Neužil and Ondra Kreibich.

My gratitude goes to all my robotics friends, who have encouraged me and cooperated with me: to Jaroslav Halgašík, Dávid Demčák, Karel Lenc, Karel Lebeda and others from team FELaaCZech; to Tomáš Novák, Monika Lacinová, Matouš Pokorný, Michal Vokáč, Marek Peca, Sean Bastable and particularly to Manolis

Chiou; and finally, to robots Bob and Henry for participating in my experiments.

Finally, none of my studies and travels would have been possible without my funding. The research leading to the results presented in this thesis has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement No 600623, STRANDS. Additionally, I would like to thank the organisers of The First Örebro Winter School on Artificial Intelligence and Robotics for allowing me to attend their winter school.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

A single symbol	A set	Meaning
	A	a set of actions that can be used to establish a plan, they are defined by an operator
a	$\mathcal{A}$	an action from a plan
$a_{\vdash}$	$\mathcal{A}_{\vdash}$	an action start point defined as $(a, t_{\vdash})$
$a_{\dashv}$	$\mathcal{A}_{\dashv}$	an action end point defined as $(a, t_{\dashv})$
$a_{\vdash\!\dashv}$	$\mathcal{A}_{\vdash\!\dashv}$	any action point
$a_{\perp}$	$\mathcal{A}_{\perp}$	an instantaneous action defined as $(a, 0)$
a_f^+	$\mathcal{A}^+$	an achiever of atom f / a set of all achievers
a_f^-	$\mathcal{A}^-$	a deleter of atom f / a set of all deleters
$a_{\vdash\!\dashv}^{can}$	$\mathcal{A}_{\vdash\!\dashv}^{can}$	a candidate action point that can be merged
	$\mathcal{A}_s^{can}$	a set of candidates that could be merged in the state S
$a_{\vdash\!\dashv}^{app}$	$\mathcal{A}_{\vdash\!\dashv}^{app}$	an action point that is applicable in the current state
$a_{\vdash\!\dashv}^{inapp}$	$\mathcal{A}_{\vdash\!\dashv}^{inapp}$	an action point that is inapplicable in the current state
	$\mathcal{B}$	a set of bindings
d		a deadline
D		a domain
$\mathcal{E}(l_i^e, l_j^s)$		an estimate of travel time
$eff(a)$		an effect of an action a
f	F	an atom
	F^+	atoms holding in the current state S
	F^-	atoms not holding in the current state S
G	$\mathcal{G}$	a goal state of a planning problem
i		an index of a task in a set
I		an initial state of a planning problem
j		an index of a task in a set
ℓ	$\mathcal{L}$	a causal link
l	L	a literal
	L_S	a set of literals currently holding in state S
l_i^s, l_i^e		a start or an end location of a task
n		an amount of tasks
ω	Ω	a task
ω_Ω		a single task consisting of all tasks from set Ω
$\omega^\times$	$\Omega^\times$	a task with a relaxed initial state
o	$\mathcal{O}$	an ordering

A single symbol	A set	Meaning
p		duration of an action or a task
P		a planning problem
P^{join}		a planning problem solved by a joining plan
$pre(a)$		a precondition of an action a
π	Π	a plan
π_i	Π_ω	a plan how to achieve a task ω_i
π_f		a final plan
π^{join}	Π^{join}	a joining plan, see Def. 12.3.8
π_s		a plan how to achieve state S
$\pi_i^\times$	$\Pi^\times$	a plan solving task $\omega_i^\times$
r		a release date
S		a search state during merging
t_s		a time stamp of state S
X		a temporal planner
γ_G		a goal related flaw
γ_A		an action related flaw

CHAPTER 1

INTRODUCTION

The first robot which was able to not only act within its environment, but also *deliberate* about its actions, was the general-purpose mobile robot Shakey (N. J. Nilsson, 1984). Despite Shakey being introduced more than 50 years ago, the world has seen very little deployment of deliberation techniques onto physical systems that would autonomously act in the real world. One reason behind this is the following phenomena. Deliberation algorithms (e.g., planners) utilise a model of the real world in order to determine a plan for how to act in the environment. For the plan to be successfully performed in the environment, the modelling language must be able to express all the necessary properties of the environment. Shakey's success was that it operated only in a simplified world that could be modelled by the proposed STRIPS modelling language. STRIPS allows to model causal relationships between actions, but it does not support modelling of temporal relations between them. Moreover, STRIPS does not support other features that may be critical for a real world deployment. For example, a plan may fail due to a shortage of energy. To summarise, the expressibility of a modelling language is a factor that limits the deployment of real systems, such as robots, in the real world.

Modern modelling languages have overcome some of these limitations. We see a variety of languages that support temporal planning, planning under uncertainty, planning with continuous resources, etc. Because the modelling languages have more expressive power, the search space of planners utilising them becomes larger and possibly harder to structure. Therefore, planners utilising these languages may have limited scalability and slow computational speed. This is the second aspect that limits the deployment of real systems with deliberation in the real world. Therefore, the motivation of this thesis is how scalability and performance of deliberation techniques can be improved.

One of the two techniques we study is temporal planning. We focus on temporal planning because it is critical for mobile service robot applications, in which we are interested. For example, imagine a user giving the task "*Come to my office at 10:00 and collect a letter*" to a robot. The task might not be valid after this time, and if the robot fails to respect this time constraint the user would have to do the task themselves.

Modern temporal planners, such as the CHIMP (Stock, Mansouri, Pecora and Hertzberg, 2015) and the FAPE (Bit-Monnot, 2016), exploit *hierarchical planning* in order to increase their scalability. In hierarchical planning, a human predefines how a *task* can be decomposed into actions to satisfy its goal. This saves planning effort because the planner does not have to find the sequence of actions by itself; it only expands the task into the actions. The idea of hierarchical planning can be generalised in the following way. The predefined tasks will become atomic, i.e., they will not be decomposed, and a deliberation system will decide only on their order. Such an approach is taken by *scheduling* techniques, which are the second deliberation techniques that we study in this thesis.

Scheduling addresses the temporal constraints imposed on tasks. The constraints limit the occurrence of a task in time by specifying a time window. Time windows are typically specified by a *release date* (the earliest time when a task can start) and a *deadline* (the latest time when a task must end). Besides the temporal constraints, physical systems (such as robots) are also restricted by spatial constraints, such as tasks are required at different locations or a robot can transfer only one object at a single time instance, etc. These constraints can be either static or dynamic. Considering dynamic spatial constraints is very important for the deployment of a real system. For example, a robot's travel between two locations can consume very different amounts of time depending on the time of day because the environment changes as a result of human activities.

In summary, this thesis addresses scheduling and temporal planning techniques in order to not only increase their scalability, but also to better accommodate temporal and spatial constraints. Our main motivation is the application of these deliberation techniques onto a mobile service robot. Nevertheless, the developed techniques are not limited to this application. The addressed problem can be illustrated by the following example, which is going to be used throughout this thesis.

1.1 Example Scenario

Consider a mobile service robot operating in an office building. It autonomously performs tasks to assist the occupants in their everyday activities. Human activities are largely based on schedules, e.g. they need to be at work, at meetings, at the airport to take a flight for a holiday, etc., at specific times. Therefore, an assistant robot must respect the given time constraints of tasks in order to satisfy humans' expectations. One can imagine a wide array of tasks for such a robot to perform. For example, some tasks may be given on a daily basis in a *routine*:

- ω_1 : "Notify people in a common eating area to leave between 20:00 - 20:15."
- ω_2 : "Check if the emergency exits are clear every 2 hours."

Other tasks, so called *on-demand* tasks, are requested by the occupants any time during a day:

- ω_3 : "Come to the reception now, pick up the letters and distribute them during the rest of the day."

- ω_4 : "Get John a cereal bar from a vending machine and bring it to him before 14:00."

Additionally, tasks are associated with different locations. In this example, the robot navigates on a topological map, see Fig. 1.1, which is represented as a *weighted graph*. Each location where the robot can perform a task is a node and a directed weighted edge from one node to another represents that the robot can travel between these nodes. The weight of an edge corresponds to how many time units the journey would consume if it were travelled by the robot. The travel in the opposite direction may consume a different amount of time. Moreover, these travel times are often affected by people's activities. As a result, the travel time between two locations are dependent on the time of day. For example, the travel

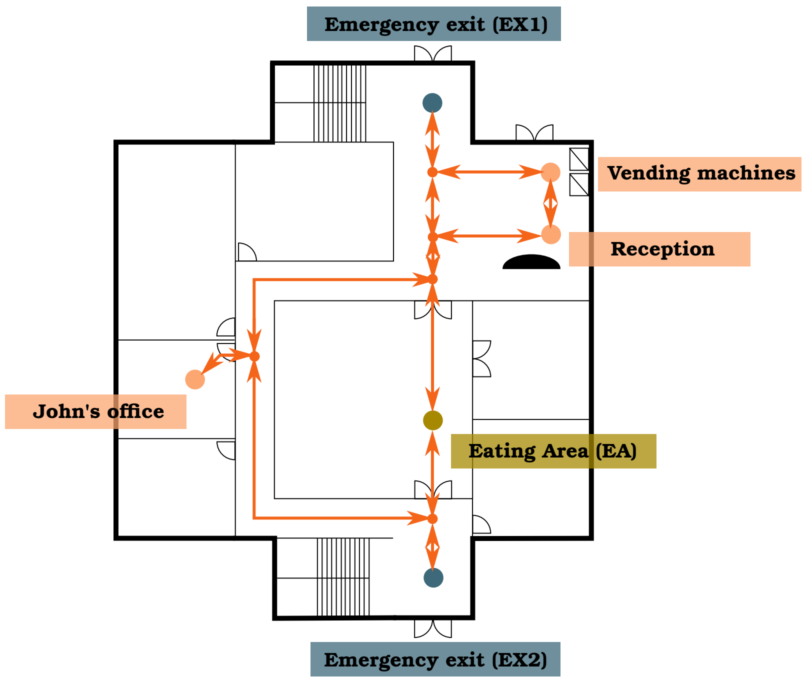


Figure 1.1: An example of a building layout and the corresponding topological map, where edge weights are ignored for clarity of the picture. Additionally, highlighted nodes refer to the locations needed by the introduced tasks.

through the common eating area may be significantly faster in the morning than at a lunch time. As a result, deliberation techniques should consider such a property in order to optimise the robot behaviour.

1.2 Scope of the Thesis

Our motivation is to develop deliberation techniques that will support a system operating in the real world, such as a mobile service robot. However, our developed techniques are general enough so that they can be deployed on any system obeying the following assumptions.

A1 All *tasks* are performed by a single agent only.

A2 The state is fully observable and actions have deterministic effects.

A3 Resources required by a task are not considered.

A4 Tasks are *separable*. A task is separable if there exists no other task that must precede it or follow it, i.e. there is no ordering constraint between tasks. This is typically the case of tasks required by different users. Inseparability can be illustrated on the case when a customer orders a starter and a main. The preparation of two meals can be viewed as two tasks. However, the task related to the starter must be finished before the main.

A5 The tasks are restricted in time by their *time windows*. Each task must be performed strictly within its window.

A6 Tasks are restricted by spatial constraints.

A7 The spatial and temporal constraints related to two tasks may interact. In the literature, this is recognised as *dependency* between tasks. Note that *separability* does not mean the same as *independence* between tasks. Independence is stronger. For example, bringing a bar to John is *dependent on* picking up letters from the reception, as the robot cannot be at different locations at the same time. Hence, these tasks needs to be ordered in a *sequence* due to their dependency. However, carrying the bar while carrying the letters is *independent*, i.e., these tasks can be ordered in *parallel*.

A8 Tasks are given in sets (referred to as *batches*).

While assumptions 1-3 limit the addressed problem, the assumptions 5-7 extend it. The fourth assumption could be avoided by modelling inseparable tasks as one bigger task. Hence, we argue that it does not really limit our problem.

The interactions between constraints related to different tasks can be classified into "*synergies*" and "*demands*". Synergies refers to interactions, which can be exploited in a plan. For example, the locations visited while executing task ω_i might also be of use for executing task ω_j. In mobile robot domains, travel time between locations is the biggest contributor to the cost of plans. Therefore, we are interested in developing a system that will exploit as many synergies as possible. Such a system would minimise the cost of plans. In contrast, demands refers to interactions, which must be obeyed. For example, there is a demand caused by conflicting spatial constraints of task ω_1 and ω_2. As each task is allocated at different location, tasks need to be ordered as the robot cannot be at the two different places at the same time.

Finally, processing tasks in batches (the eighth assumption) is one possible technique that allows planning for sets of tasks. Another technique is a *continuous* approach where a plan is made for the first task and then changed to accommodate the second task, etc. Because batch processing dominates scheduling and temporal planning, we follow it as well. However, deploying batch scheduling or planning onto a real robot raises one important issue. As the robot keeps receiving new tasks, it needs to obtain schedule/plan for a new batch either after the previous batch is executed or shortly before its end. In both cases, the robot's performance can be optimised if there is an estimate on how long a scheduler/-planner needs to produce the new schedule/plan. Therefore, a relevant property of a scheduler or a planner is how deterministically it behaves on the given problems. We will elaborate on this property in the evaluation.

1.3 Contributions and Related Publications

This thesis is split into two parts. The first part focuses on scheduling. Thus, it adds one more assumption that a task is represented as a single action to perform. As a result, it reasons only about *when* a task should be executed in order to satisfy all spatial and temporal constraints. Scheduling is traditionally modelled either as a Constraint Satisfaction Problem (CSP) or as a Mixed Integer Program (MIP). In our work, we build on the MIP formalisation as it naturally supports the notion of optimising the found solution, i.e., finding the best schedule. Particularly, we improve on a scheduler proposed by (Coltin, Veloso & Ventura, 2011), where our contributions are as follows.

1. We propose a heuristic based on Allen's interval algebra to prune the search to be performed by a MIP. We demonstrate how this heuristic significantly speeds up the developed scheduler and we contrast these results with performance of Coltin's et al.'s scheduler. This work has been published in (Mudrová and Hawes, 2015).

2. Similarly to Coltin's et al.'s scheduler, which is deployed on mobile robots called CoBots, our proposed algorithm has been used on several mobile robots in long-term autonomy scenarios, where it schedules large sets containing a variety of tasks. Additionally, the scheduler is open-source[1].

[1]https://github.com/strands-project/strands_executive

3. The developed scheduler is also integrated into a control framework for the long-term task-driven control of mobile service robots. We illustrate how the scheduler adjusts schedules during a day due to the impact of this framework. This framework has been developed in cooperation with Dr. Bruno Lacerda and Dr. Nick Hawes. It has been published in (Mudrová, Lacerda and Hawes, 2015).

The second part of this thesis focuses on *temporal planning*. Therefore, our proposed deliberation system does not only tackle *when* to perform a task but also *how*. Exploiting the assumption that tasks are separable, our developed system is based on *merging* of separately generated plans for each task. Our merging algorithm builds on plan coordination which is well known in multi-agent domains. Our contributions are as follows.

4 We propose a partial order temporal plan merging algorithm POPMER$_X$ that is able to generate a plan for a large number of tasks while taking advantage of possible *synergies* and respecting *demands* between such tasks. We demonstrate how this algorithm performs very well in a comparison to the state-of-the-art planners, such as OPTIC (Benton, Coles & Coles, 2012), TFD (Eyerich, Mattmüller & Röger, 2009) and YAHSP (Vidal, 2004). This work has been published in (Mudrová, Lacerda & Hawes, 2016).

5 POPMER$_X$ supports task time windows while exploiting state-based planning. This expands the current state-of-the-art as PDDL 3.0 (A. Gerevini & Long, 2005) supports modelling of deadlines but not of the release dates. Modelling of release dates is possible in PDDL 2.2 (Edelkamp & Hoffmann, 2004) but it is very cumbersome.

6 We have released implementation of POPMER$_X$ and all testing domains as open source[1].

[1]https://github.com/mudrole1/POPMER

Part I

Scheduling

CHAPTER 2

INTRODUCTION

We are interested in a scheduler which receives as an input a set of tasks and finds the time instants when the execution of those tasks should be started while respecting given constraints. We build on the terminology from the classical scheduling of *jobs* on machines in factories (Pinedo, 2012). However, we refer to a *task* instead of a *job*, in order to unify this terminology with the rest of the thesis. We build on the classical scheduling in order to unify the nomenclature from literature focusing on scheduling for robots, as it strongly varies. For example, the term "scheduling" is often misused to solely describe the *ordering* of tasks or to express the fact that a task has *time properties* (whether or not their execution is organised with reference to these properties). For example, ordering appeared (as scheduling) for the robot RHINO (Beetz and Bennewitz, 1998) and expressing time constraints appeared (again as scheduling) in the robot MINERVA (Thrun et al., 1999), extending work from (Burgard et al., 1998).

In classical scheduling, a task requires to be a process on a machine with additional resources. For example, a tool, a person or a certain level of energy may be modelled as a resource for the task. In the scheduling community, three different general scheduling problems are recognised based on how they address the resources required by a task (Heinz & Beck, 2011): task scheduling (TS), resource allocation (RA) and joint scheduling (TSRA). TS decides *when* a task should be executed, i.e., it finds such a sequence of tasks so that all tasks can access their required resources whilst the capacities of the resources are not violated. In contrast, RA focuses on a situation where tasks are already scheduled and dynamic resources need to be assigned to the tasks. Finally, TSRA combines both of the above problems. We focus on the task scheduling problem as the only considered resource for a task is a single agent (due to assumptions **A1** and **A3**). From now on, we will refer to this problem only as "scheduling".

Heinz and Beck (2011) defines the general scheduling as: "*Given a set of tasks that require the use of one resource from a set of alternative resources, a solution will assign each task to a resource and schedule the tasks such that the capacity of each resource is respected at all time points.*" However, this definition is incomplete because a given scheduling problem can have more than one solution. Thus, a scheduler also needs to solve an optimisation problem using an objective criterion to choose one solution. Such criterion can be minimum makespan, which represents the requirement that all the tasks are finished as soon as possible with minimal spaces between them.

Such a scheduling problem is formalised as a set of three parameters (Pinedo, 2012). The first parameter describes a machine environment, the second specifies the constraints and processing details to be obeyed and the third defines an optimisation criterion.

The assumptions **A1** - **A8** translate into the scheduling problem as follows. The problem is constrained to a *single machine* environment due to the assumption **A1**, which requires that all tasks can be executed by a single agent. On one hand, the assumptions **A2** - **A4** simplify the processing details. Hence, the scheduler neither allocates resources nor preserve ordering on tasks. On the other hand, processing details must support that tasks are strictly performed in time windows (**A5**) and the schedule considers spatial constraints as well (**A6**). The dependency between tasks (**A7**) is recognised by a scheduling technique while optimising the solution according to a chosen criterion. Finally, the scheduling problem is *static*, as the set of tasks is known beforehand (**A8**) (Bidot, Vidal, Laborie and Beck, 2009). The solution to this problem is to assign to each task its start of execution in a way that no tasks overlap (the capacity of the single agent is respected) and all time windows are respected.

2.1 Additional Assumptions

In addition to the main assumptions, scheduling adds the following assumption.

A9 A task is a predefined inseparable sequence of steps.

This means that a task needs to be executed as a whole. Therefore, the scheduler treats tasks as *black-boxes*, which means that it does not reason about tasks' steps.

Furthermore, we set the following requirements for a scheduler based on the demands of our case scenario. The speed of the algorithm is generally more important than the optimality of a final schedule, as we prefer the robot to act within a reasonable time limit when given a task. This is motivated by the fact that users need to wait for their tasks to be performed and the slow performance of the robot may decrease their interest. Therefore, we set a limit for the scheduler that it must provide a solution within three minutes.

2.2 Example of Scheduling with Spatial Constraints

Consider a robot that receives two tasks ω_1: "notify people in a common eating area to leave between 20:00 - 20:15" and ω_2: "check if the emergency exits are clear every 2 hours" from the case scenario. We will discuss later how the periodic property of task ω_2 can be handled. For now, let us assume that the task is required

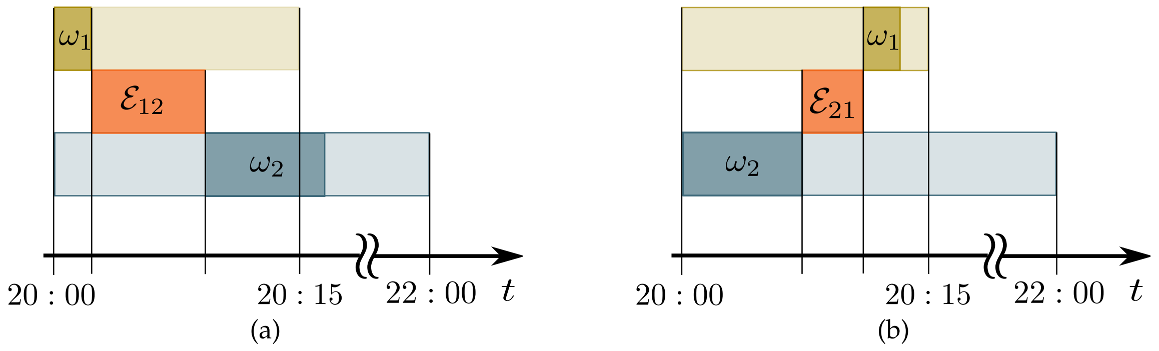


Figure 2.1: Two schedule possibilities: either (a) ω_1 precedes ω_2 or (b) ω_2 precedes ω_1. The estimates $\mathcal{E}_{12}, \mathcal{E}_{21}$ of travel times between the eating area and the first emergency exit may significantly differ based on the order of tasks.

to happen between 20:00 - 22:00 as a result of its periodicity. Task ω_1 is associated with the location "Eating Area (EA)". Similarly, task ω_2 is associated with two locations EX1 and EX2, which are related to the two emergency exits. Therefore, the robot must navigate to these locations first before the corresponding task can be performed.

These spatial constraints can be reflected by ensuring that there is enough time between two tasks to navigate between their locations. Moreover, this time is not constant in the mobile robot domain. Recall that the travel time between two locations may be dependent not only on the direction but also on time during the day, as discussed in the example scenario description. Fig. 2.1 illustrates this situation where the travel time between two tasks ω_1 and ω_2 differs based on their order. To conclude, we are going to describe the proposed scheduler that considers such time-dependent spatial constraints.

2.3 Organisation

This part of the thesis is organised as follows. We first give a review of related work (Chapter 3). In our review, we focus on the literature related to robots performing tasks with temporal constraints. Because scheduling is a very broad field, we refer the interested reader to a comprehensive overview given in (Pinedo, 2012). In Chapter 4, we formalise the introduced scheduling problem and related terminology. Then, we introduce our proposed scheduler in Chapter 5. The evaluation in Chapter 6 contrasts this scheduler with a state-of-the-art scheduler (Coltin et al., 2011). We first compare their properties on simulated data in order to observe how different dataset properties influence the schedulers behaviour. Furthermore, as our scheduler was deployed on a number of robots in long-term scenarios, we compare the schedulers on the data collected during these deployments. We conclude by discussing the advantages and disadvantages of our approach.

CHAPTER 3

RELATED WORK

The boundary between scheduling and temporal planning is unclear (Smith, Frank & Jónsson, 2000), as both fields overlap. Therefore, we review and analyse different techniques from both disciplines which are able to address the problem considering assumptions **A1** - **A9**. In the review, we focus on how these techniques support task execution on mobile robots operating in the real world.

3.1 Scheduling

The introduced case scenario can be simplified into different sub-problems that can be solved by techniques related to *Vehicle Routing Problem* (VRP). The full scenario can be addressed either by *Mixed-Integer Programming* (MIP) or as a *Constraint Satisfaction Problem* (CSP).

3.1.1 Vehicle Routing Problem

VRP focuses only on optimising the travel cost of robots traversing between different locations. Therefore, the tasks are neglected. Furthermore, VRP can be extended into a *Pick up and Delivery Problem* (PDP) (sometimes also refered to as a *Collection and Delivery Problem*), which assumes only one type of a task: an object delivery. In this problem, a robot needs to pick up an object in one location and deliver it to another. PDP extends the VRP problem by adding two constraints:

C1 the same robot visits the pick up and delivery destination;

C2 the robot has a limited capacity in how many objects it can transfer.

Furthermore, two sub-problems are recognised: with or without preemption. In the former case, a robot can be interrupted to collect another item as long as its capacity allows. In the latter case, the robot must first deliver the current object before picking up another. Finally, VRP and PDP is also extended to handle Time Windows (VRP-TW, PDP-TW), where robots must visit the locations within specified time windows, as in our example scenario. The preemptive PDP-TW with limited capacity is addressed by the ARIADNE robots (Surmann & Morales, 2002).

Coltin and M. Veloso (2014) further extend the PDP to accommodate the possible *transfer* of the picked up object onto another robot which will finish the delivery (PDP-T). Hence, the constraint C1 is omitted. The motivation behind such problem is that the delivery times and cost may be improved by dividing the labour among several robots.

The PDP-TW with preemption is further extended into *Flexible Manufacturing System* (FMS). In FMS, a fleet of mobile robots must ensure that static machines have enough material to process. Additionally, the robots may perform some manufacturing themselves. However, these tasks can be *preempted* when needed because the transformation of the material has higher priority than the manufacturing. The goal is to minimise the makespan of the resulting schedule. Finally, FMS can be solved using either MIP techniques (Nielsen, Dang, Nielsen & Pawlewski, 2014) or via a heuristic approach based on an genetic algorithm (GA) (Dang & Nguyen, 2016).

3.1.2 General Task Scheduling

Mixed-Integer Programming (MIP) and *Constraint Satisfaction Problems* (CSPs) are popular techniques to model scheduling problems. In particular, *Simple Temporal Networks* (STNs) (Dechter, Meiri & Pearl, 1991) are very popular network-based methods to solve scheduling CSPs. The main difference between MIP and CSP is in how they deal with the combinatorial explosion (Booth, Tran, Nejat & Beck, 2016). Booth et al. (2016) have compared both techniques in a scenario where robot Tangy interacts with humans in order to schedule games for them. They have proven that both methods are appropriate approaches to use for scheduling on mobile service robots. Additionally, they have illustrated that both techniques are faster than the previous version of Tangy's controller (Louie, Vaquero, Nejat & Beck, 2014) that has exploited the OPTIC planner (Benton et al., 2012).

Additionally, STNs are used in a robot which monitors the activities of an elderly person (Cesta et al., 2011). The activities and their time relations are prepared by a carer beforehand and they are mapped to a temporal constraint language in O-Oscar architecture (Cesta, Cortellessa, Oddi, Policella & Susi, 2001), which is built on top of an STN. Furthermore, Coltin et al. (2011) have proposed a scheduler based on MIP for the CoBots project (M. M. Veloso et al., 2012), where multiple indoor robots operate in an office building.

MIP is also used for multi-robot missions, where a fleet of robots needs to satisfy tasks by cooperating (Korsah, Kannan, Browning, Stentz & Dias, 2012). As a result, their schedules need to be synchronised. Additionally, MIP can be also exploited to handle detailed spatial constraints while allocating tasks to different robots. For example, a fleet of robots is scheduled to patrol regions of interest (Atay, 2007) while scheduling has considered the physical limitations imposed by the environment. Especially in such scenarios, it may be beneficial to run a scheduler longer in order to obtain a more precise solution. Therefore, Li, Sweeney, Ramamritham, Grupen and Shenoy (2003) present a scheduler that returns different feasible solutions depending on the given resources, such as time to operate.

3.2 Planning

Planning would represent the scheduling problem by representing tasks as actions. Chen, Jin and Yang (2012) recognise three categories of how actions can be modelled. The two most related to us are:

- actions are specified by their preconditions and effects, as in PDDL;
- actions are abstractions, as in Hierarchical Task Networks (HTNs).

While the first category focuses on the support of general automated planning, the second category provides the abstractions in order to improve search efficiency. We review several robot applications for both categories. Nevertheless, not all reviewed techniques support temporal planning. However, they are still very relevant to us as they address very similar scenarios to ours.

3.2.1 Action Description Languages

The STRIPS language (Fikes & Nils J. Nilsson, 1971) was the first action description language (ADL) used to model problems for the robot Shakey (N. J. Nilsson, 1984). STRIPS was further extended into PDDL (M. Ghallab et al., 1998), $\mathcal{B}$ and $\mathcal{C}$ languages (Gelfond & Lifschitz, 1998). The language $\mathcal{C}$ was further extended into $\mathcal{C}+$ (Giunchiglia, Lee, Lifschitz, McCain & Turner, 2004), which provides easy-to-follow concepts on how to model some important features for task execution, such as concurrent or non-deterministic actions. Furthermore, $\mathcal{BC}$ language combines features of $\mathcal{B}$ and $\mathcal{C}+$ (Lee, Lifschitz & Yang, 2013). As a result, $\mathcal{BC}$ language supports indirect and recursive effects of actions. $\mathcal{C}+$ and $\mathcal{BC}$ languages are widely used to model mobile service robot scenarios. Additionally, their advantage is that they can be translated into an answer set programming problem (ASP) and solved by existing ASP solvers.

3.2.1.1. PDDL

As mentioned previously, the social robot Tangy schedules multi-user activities, while considering users' schedules (Louie et al., 2014). The robot performs only two tasks – performing the game Bingo for more users and reminding a single user about an upcoming game. It uses the OPTIC temporal planner (Benton et al., 2012) that uses PDDL to specify a set of tasks and users' schedules are represented as Timed Initial Literals (TILs) (A. E. Gerevini, Haslum, Long, Saetti & Dimopoulos, 2009).

Furthermore, the ROSPlan (Cashmore et al., 2015) is a general framework that allows for a task planner to be embedded into the Robot Operating System (ROS), a middleware widely used in the robotics community. As a proof of concept, ROSPlan has integrated the POPF planner (A. J. Coles, Coles, Fox & Long, 2010), an ancestor of OPTIC.

3.2.1.2. $\mathcal{C}+$

The service robot Ke Jia (Chen et al., 2010) receives commands to pick and deliver some household objects. Another household scenario is described in (Aker, Erdogan, Erdem & Patoglu, 2011), where a robot performs housekeeping tasks. Furthermore, Havur, Haspalamutgil, Palaz, Erdem and Patoglu (2013) present how $\mathcal{C}+$ can be used to address the *Tower of Hanoi puzzle*, which has been proposed as a robotics challenge as a part of EU Robotics coordination action in 2011. Because a problem described by $\mathcal{C}+$ can be transformed into a SAT problem by the reasoner CCALC (McCain and Turner, 1997), Havur et al. (2013) have compared

$\mathcal{C}+$ approaches with SAT (SATisfiability) approaches. Additionally, $\mathcal{C}+$ language can also be used to control Lego Mindstorms robots, as shown in (Caldiran et al., 2009).

3.2.1.3. $\mathcal{BC}$

Khandelwal, Yang, Leonetti, Lifschitz and Stone (2014) illustrate how $\mathcal{BC}$ language can support planning with incomplete information. It exploits the feature that a problem in $\mathcal{BC}$ can be translated into a problem of *Answer Set Programming* (ASP). They build on the previous work of Erdem, Aker and Patoglu (2012). Erdem et al. (2012) have extended the description of actions in ASP in order to handle not only geometric constraints but also temporal reasoning about durations of actions. While their work finds the shortest plan to satisfy the goal within pre-specified time constraints, the planner proposed by Khandelwal et al. (2017) also considers the cost of each action. As a result, it optimises the robot performance. This planner has been deployed on BWIBots in mobile service robot scenarios.

3.2.2 Hierarchical Task Planning

Dvořák, Bit-Monnot, Ingrand and Malik Ghallab (2014) proposed an environment FAPE, which is based on hierarchical task planning. It closely integrates planning and execution on robots, which has been demonstrated on the PR2 robot in a housekeeping scenario. Very similar concepts are exploited in the planner CHIMP (Stock et al., 2015). This planner builds on the Meta-CSP planner (Rocco, Pecora & Saffiotti, 2013a), which has been deployed on robots in a senior residential facility. The resulting problem contains causal, temporal, resource and information constraints, which are encoded as a configuration plan. None of the aforementioned approaches consider space constraints on objects. This is overcome in

(Mansouri & Pecora, 2014b).

3.3 Summary

The work of (Booth et al., 2016) on evaluating MIP and CSP in comparison to the use of the OPTIC planner on robot Tangy is very important. As the authors illustrated, MIP and CSP solved the problem significantly faster than OPTIC. Nevertheless, this should not be a surprise. Planners, in general, address more complex problems than just scheduling. Hence, their performance may be unnecessarily cumbersome on simple problems. Additionally, planning is not required in this part of the thesis due to assumption A9, which guarantees that there is already a valid sequence of steps to perform for each task. As a result, we focus on scheduling techniques, such as MIP techniques, because they are more widespread than CSPs in mobile robot scenarios.

CHAPTER 4

PROBLEM DEFINITION

We are interested in a scheduler which receives as input a set of tasks and finds the time instants when the execution of those tasks should be started. Additionally, the imposed ordering on the tasks should minimise an optimisation criterion. Furthermore, as only the case with a single robot is considered, the scheduler must ensure that no tasks overlap.

4.1 Task

A *task* represents an instance of a behaviour that the robot should perform. We refer to a single task by ω with numeric subscript i, for example, ω_1. Its properties – listed below – are then referred by the same subscript.

4.1.1 Standard Task

A standard task ω_i is defined by three aspects: an *activity*, *temporal constraints* and *spatial constraints*. The activity consists of a predefined sequence of steps π_i to perform. Temporal constraints reflect the fact that the task can start at *release date* r_i or later, and it must end before or at *deadline* d_i. Thus, the *time window* $\langle r_i,\, d_i \rangle$ is defined for each task. The sequence of steps π_i can start at any time instant s_i

within this time window, but it must finish at time e_i which is before or at the deadline. Furthermore, the task lasts for a certain amount of time, denoted as *processing time* p_i. As a result, the following equation must hold:

$$s_i,\, e_i \in \langle r_i,\, d_i \rangle \,\wedge\, s_i + p_i = e_i. \tag{4.1}$$

Finally, the spatial constraints reflect that a robot starts to execute the task at location l_i^s and ends at location l_i^e. Therefore, there may be a time gap between execution of two consecutive tasks because a robot must travel to different locations. This gap has varying duration $\mathcal{E}(l_i^e, l_j^s)$ dependent on the corresponding locations. Finally, the task can be summarised as follows.

Definition 4.1.1. ***Standard task:***
A task is defined as a tuple $\omega_i = \langle(\pi_i), (p_i, r_i, d_i, s_i, e_i), (l_i^s, l_i^e)\rangle$.

4.1.2 Extensions

Additionally, our example scenario requires *on-demand* tasks. Because these tasks need to be executed immediately, they do not have associated time window. Moreover, these tasks are not passed into the scheduler; they are executed immediately.

Definition 4.1.2. ***On-demand task:***
An on-demand task is defined as a tuple $\omega_i = \langle(\pi_i), (p_i, s_i, e_i), (l_i^s, l_i^e)\rangle$.

Furthermore, standard tasks can also have different priorities $\psi_i \in \mathbb{N}$. Priorities are especially useful to deal with an oversubscribed situation, where a robot cannot execute all the required tasks. Hence, some tasks, usually the ones with the lowest priorities, are not performed. Additionally, tasks can also be preempted. Therefore, we introduce an extended task.

Definition 4.1.3. ***Extended task:***

An extended task is defined as a tuple $\omega_i = \langle(\pi_i, \psi_i, \iota_i), (p_i, s_i, e_i), (l_i^s, l_i^e)\rangle$, where a boolean flag ι_i signals if a task execution can be interrupted or not.

4.2 Scheduling as an Optimisation problem

Scheduling is formalised as an optimisation problem. This means that a scheduler tries to find the best schedule according to the given optimisation criterion. In our case scenario, we especially want to achieve a high utilisation of the robot. Hence, the best schedule should be as short as possible. This can be expressed as minimising the schedule's *makespan*. The makespan is defined as the latest completion time, i.e. $\max(e_1, \ldots, e_n)$, where n is the number of tasks. However, we argue that the minimisation of the makespan cannot be used as the optimisation criterion because it would minimise only the end time of the last task. This does not necessary limit the other tasks, especially in the situation when the task time windows do not overlap. Therefore, we minimise *total completion time* criterion $\sum_{i\in\{1\ldots n\}} e_i$. This criterion ensures that *all* tasks end as soon as possible.

4.2.1 Problem Statement

Following the standard terminology (Pinedo, 2012), a scheduling problem is formalised by three parameters $\alpha \,|\, \beta \,|\, \gamma$, where α describes a machine environment, β specifies the constraints and processing details to be obeyed and γ defines an optimisation criterion. The symbol | does not have any special meaning, it only separates parameters into these three groups. Hence, the problem reflecting the assumptions A1 - A9 can be formalised as the following scheduling problem. Note that only standard tasks are supported within this problem.

$$1|\, r, setting_{ij}, d \,| \sum_{i \in \{1 \ldots n\}} e_i \,, \tag{4.2}$$

where

- 1 refers to scheduling for a single robot;
- r restricts that a scheduler must obey tasks' *release dates;*
- $setting_{ij}$ stands for *setting time* needed between processing of task ω_i and task ω_j.
- d refers to the fact that deadlines must be respected.
- $\sum_{i \in \{1 \ldots n\}} e_i$ is the optimisation criterion referred as the *total completion time*.

Scheduling for a single robot implicitly includes the constraint that scheduled tasks cannot overlap. Additionally, the time $setting_{ij}$ corresponds to the temporal gap between two tasks ω_i and ω_j. This gap is needed by a robot to travel from the end location l_i^e of task ω_i to the start location l_j^s of task ω_j, i.e., $setting_{ij} = \mathcal{E}(l_i^e,\, l_j^s)$. Note that a robot may take a different amount of time to travel from l_i^e to l_j^s than in the reverse direction, due to environmental constraints. A solution to this scheduling problem assigns a start time instance s_i to each input task obeying its time constraints.

CHAPTER 5

SOLUTION

In this chapter, we build on the scheduler proposed by Coltin et al. (2011). We have chosen this scheduler because it has been used extensively in experiments in real environments and it reliably solves scheduling problems in a scenario similar to ours – fulfilling users' tasks in large office buildings. It also satisfies our requirements for release dates, deadlines and it considers travel times between locations.

In contrast to this work, we contribute the three following extensions:

- Our proposed scheduler significantly increases the scalability of Coltin's et al.'s scheduler. This is achieved by heuristically pruning the search space.
- We have developed a control framework in order to support execution of tasks by a real robot. This framework not only manages the incoming tasks, but also controls and monitors their execution.
- The control framework handles on-demand tasks required by our example scenario and extended tasks as well. The extended tasks are proposed in order to allow an *oversubscribed* robot. Therefore, priorities and interruptibility of tasks are used in order to decide which tasks should be executed by the robot and which should be withdrawn.

5.1 Scheduler

We address the optimisation problem defined in Eq. 4.2 by Mixed-Integer Programming (MIP), similarly to Coltin et al. (2011). MIP is intended to be used as an optimisation technique for problems where variables have continuous and integer domains. Importantly, the variables are limited with linear constraints and the optimisation criterion is linear as well. In our work, we use the existing solver SCIP (Achterberg, 2009). Finally, the scheduling problem can be reformulated into the MIP problem:

$$\min \sum_{i \in \{1 \dots n\}} e_i \tag{5.1}$$

$$\text{subject to: } e_i = s_i + p_i \qquad \forall i \tag{5.2}$$

$$s_i \in \langle r_i, d_i - p_i \rangle \qquad \forall i \tag{5.3}$$

$$s_i + p_i + \mathcal{E}(l_i^e, l_j^s) \leq s_j + A \cdot pre_{ij} \qquad \forall i,j \tag{5.4}$$

$$s_j + p_j + \mathcal{E}(l_j^e, l_i^s) \leq s_i + B \cdot (1 - pre_{ij}) \qquad \forall i,j \tag{5.5}$$

$$pre_{ij} \in \{0, 1\} \qquad \forall i,j \tag{5.6}$$

$$A = d_i + \mathcal{E}(l_i^e, l_j^s) - r_j \qquad \forall i,j \tag{5.7}$$

$$B = d_j + \mathcal{E}(l_j^e, l_i^s) - r_i \qquad \forall i,j \tag{5.8}$$

These constraints ensure that all the time properties of the tasks are fulfilled (Eq. (5.2) and Eq. (5.3)) and that no two tasks overlap (Eq. (5.4) and Eq. (5.5)). These equations have been produced by the *Big M method* (Griva, Nash & Sofer, 2008).

5.1.1 Big M method

In general, two orderings are possible for a pair of tasks ω_i, ω_j, either "ω_i precedes ω_j" *or* "ω_j precedes ω_i". This can be modelled as a disjunction of two constraints:

$$s_i + p_i + \mathcal{E}(l_i^e, l_j^s) \leq s_j$$
$$\vee \tag{5.9}$$
$$s_j + p_j + \mathcal{E}(l_j^e, l_i^s) \leq s_i$$

In the worst case, there can be $\binom{n}{2}$ combinations of disjunction pairs of constraints, where n is the number of tasks. However, the disjunction of constraints cause difficulties while solving them with MIP. Therefore, the motivation behind the Big M method is translation of Eq. 5.9 into a conjunction of two constraints.

The Big M method modifies Eq. 5.9 in the following steps. First, let us assume that task ω_i precedes task ω_j. In this case, the top equation in Eq. 5.9 would be true and the bottom one false; the overall expression would be true. If the disjunction is replaced by conjunction, the overall expression would be false. Therefore, the bottom equation needs to be modified in the way that it will be true as well. However, it should not limit the solution. Therefore, the bottom equation must become tautology in the case when ω_i precedes task ω_j. Hence, the Big M Method extends the equation to become tautology:

$$s_i + p_i + \mathcal{E}(l_i^e, l_j^s) \leq s_j$$
$$\wedge \tag{5.10}$$
$$s_j + p_j + \mathcal{E}(l_j^e, l_i^s) \leq s_i + B$$

The value of B must be big enough that the equation holds, i.e., $B \geq s_j + p_j + \mathcal{E}(l_j^e, l_i^s) - s_i$. In order to find the biggest value for B, we can observe that $s_j + p_j = e_j$. The latest time when the task can end is at its deadline, thus $e_j = d_j$ in the worst case. In contrast, the earliest time when task ω_i can start is at its release date, $s_i = r_i$. As a result,

$$B = d_j + \mathcal{E}(l_j^e, l_i^s) - r_i.$$

Second, let us assume now the opposite situation when ω_j precedes ω_i. The added constant B has invalidated the bottom equation because task ω_i could be now started very soon. Therefore, the constant B should be added only when ω_i precedes ω_j. As a result, the Big M method adds into the problem an *integer* variable

$$pre_{ij} = \begin{cases} 0 & \text{if task } \omega_i \text{ precedes task } \omega_j \\ 1 & \text{if task } \omega_j \text{ precedes task } \omega_i \end{cases}$$

and the equation is modified to:

$$\begin{gathered} s_i + p_i + \mathcal{E}(l_i^e, l_j^s) \leq s_j \\ \wedge \\ s_j + p_j + \mathcal{E}(l_j^e, l_i^s) \leq s_i + B \cdot (1 - pre_{ij}) \end{gathered} \tag{5.11}$$

This guarantee that the constant B is added only when ω_i precedes ω_j.

Finally, the similar changes are made to the top equation, which results in:

$$\begin{gathered} s_i + p_i + \mathcal{E}(l_i^e, l_j^s) \leq s_j + A \cdot pre_{ij} \\ \wedge \\ s_j + p_j + \mathcal{E}(l_j^e, l_i^s) \leq s_i + B \cdot (1 - pre_{ij}), \end{gathered} \tag{5.12}$$

where

$$A = d_i + \mathcal{E}(l_i^e, l_j^s) - r_j.$$

5.1.2 Proposed Extensions

The pair of constraints (Eq. (5.4) and Eq. (5.5)) significantly complicates the scheduling problem. We observe that these constraints are not needed in situations when two tasks do not overlap. For the situation when tasks overlap, we propose a heuristic based on *Allen's interval algebra* (Allen, 1983) to pre-select one of the possible orderings for each pair of tasks. This local optimisation greatly reduces the effort required by the MIP solver, but it does so at the cost of pruning possible solutions from the search space (losing both optimality and in some cases completeness).

Our proposed scheduler works as follows:

1. Every possible pair of tasks from the set is analysed.
2. Allen's interval algebra is used to prune the possible orderings for each pair.
3. The constraints are specified for the chosen ordering.

Step 3) is based on Coltin et al. (2011). The other steps represent the novel elements of our algorithm.

5.1.2.1. Analysis of a Pair of Tasks

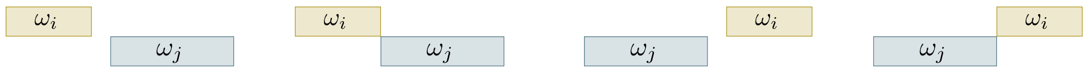
Having a pair of tasks ω_i and ω_j, four situations σ might occur:

- only "ω_j precedes ω_i" is possible (σ_0);
- only "ω_i precedes ω_j" is possible (σ_1);
- both situations are possible (σ_2);
- neither are possible (σ_3).

The situation σ for any pair of tasks depends on the relationship between their time windows, see Fig. 5.1 -Fig. 5.4. If the time windows do not overlap (relations "before" and "meets"), then determining the ordering is simple. Otherwise, we use the following process to determine which situation holds.

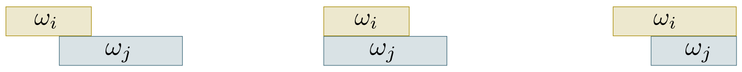
First, we compute the parameter

$$\varepsilon_o = \min(d_j - r_i, d_i - r_j).$$



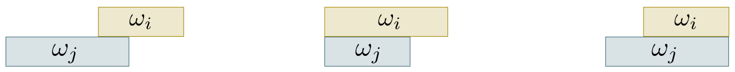
(a) ω_i before ω_j (b) ω_i meets ω_j (c) ω_j before ω_i (d) ω_j meets ω_i

Figure 5.1: For these Allen's relations of time windows, we add neither the constraint Eg. 5.4 nor Eq. 5.5.



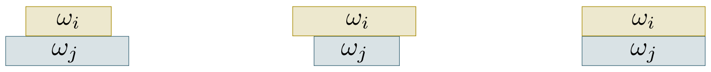
(a) ω_i overlaps ω_j (b) ω_i starts ω_j (c) ω_j finishes ω_i

Figure 5.2: For these Allen's relations of time windows, we add only constraint Eg. 5.4, i.e., that ω_i precedes ω_j.



(a) ω_j overlaps ω_i (b) ω_j starts ω_i (c) ω_i finishes ω_j

Figure 5.3: For these Allen's relations of time windows, we add only constraint Eg. 5.5, i.e., that ω_j precedes ω_i.



(a) ω_i during ω_j (b) ω_j during ω_i (c) ω_i equals ω_j

Figure 5.4: For these Allen's relations of time windows, we first compute the total completion time for both situations, i.e. either ω_i precedes ω_j or vice versa. Then, we choose the situation, which locally minimises the total completion time.

ε_0 is positive if the time windows for tasks ω_j and ω_i overlap. Then, we test, which ordering – ω_j precedes ω_i or ω_i precedes ω_j – is possible. For each possible ordering, the maximal size of an interval where both tasks can fit is $\varepsilon_1 = d_i - r_j$ and $\varepsilon_2 = d_j - r_i$, respectively. Both tasks' durations and travel time between locations must fit in these maximal intervals. Therefore, the following equations are tested:

$$\varepsilon_1 \geq p_j + \mathcal{E}(l_j^e, l_i^s) + p_i, \tag{5.13}$$

$$\varepsilon_2 \geq p_i + \mathcal{E}(l_i^e, l_j^s) + p_j. \tag{5.14}$$

Then,

- σ_0 occurs iff only Eq. (5.13) holds;
- σ_1 occurs iff only Eq. (5.14) holds;
- σ_2 occurs iff both equations hold;
- σ_3 occurs iff neither equation holds.

5.1.2.2. Pruning

When σ_2 occurs, our algorithm picks one ordering constraint – ω_j precedes ω_i or ω_i precedes ω_j – which it considers (locally) to be the best. This decision is made based on the thirteen possible relations of two intervals described by Allen's interval algebra. The situation σ_2 can appear on relations "overlaps", "starts", "finishes", "during" and "equals". Motivation for pruning can be illustrated on the following example. Assume that time window of ω_i *overlaps* the window of ω_j. Because ω_i can start sooner than ω_j and ω_j can end later than ω_i, the order ω_i precedes ω_j locally maximises the chances that the tasks fit into their time windows. This leads to locally optimising the chances to find a valid schedule. We utilise the value of ε_1 and ε_2, which are computed anyway, in order to choose which ordering is better. Hence, for relations "overlaps", "starts", "finishes", the ordering constraint is chosen by testing the formula:

$$\varepsilon_1 > \varepsilon_2.$$

If the formula is true, then ordering ω_j precedes ω_i is chosen and vice versa. This maximises the amount of time available for the tasks.

For the remaining interval relations (ω_j during ω_i, ω_i during ω_j and ω_j equals ω_i) the possible orderings are indistinguishable using the previous rule. Therefore, we choose the ordering constraint which locally minimises the global optimisation criterion $\sum e_i$. We calculate this for a pair of tasks as follows. First, we calculate the criterion $C_0 = \sum_{k\in\{i,j\}} e_k$, assuming task ω_j precedes task ω_i. As no other tasks are considered, task ω_j can start as soon as possible, followed by task ω_i. However, if that would imply that task ω_i could start before its release date, then the start of the task is set to its release date.

$$\begin{aligned} s_j &= r_j \\ s_i &= \max(r_j + p_j + \mathcal{E}(l_j^e, l_i^s), r_i). \end{aligned} \tag{5.15}$$

Then, the criterion for this ordering is:

$$C_0 = \sum_{k\in\{i,j\}} e_k = e_j + e_i = (s_j + p_j) + (s_i + p_i).$$

Using Eq. (5.15), this equation expands into:

$$C_0 = \begin{cases} (r_j + p_j) + (r_j + p_j + \mathcal{E}(l_j^e, l_i^s) + p_i), \text{ when } s_j > r_j; \\ r_j + p_j + r_i + p_i, \text{ when } s_j = r_j; \end{cases} \tag{5.16}$$

Next we calculate the criterion $C_1 = \sum_{k\in\{i,j\}} e_k$, assuming the opposite ordering – task ω_i precedes task ω_j. The criterion is similar

$$C_1 = \begin{cases} (r_i + p_i) + (r_i + p_i + \mathcal{E}(l_i^e, l_j^s) + p_j), \text{ when } s_j > r_j; \\ r_j + p_j + r_i + p_i, \text{ when } s_j = r_j; \end{cases} \tag{5.17}$$

Finally we choose the ordering which produces the lowest criterion value. If they are equal, then the order does not matter and we choose task ω_j to precede task ω_i.

5.1.2.3. Scheduler Constraints

Following the results of pruning, there is no need for integer variables any more. Hence, the MIP is simplified to a linear program (LP) defined as:

$$\min \sum_{i \in \{1 \ldots n\}} e_i \tag{5.18}$$

$$\text{subject to: } e_i = s_i + p_i \qquad \forall i \tag{5.19}$$

$$s_i \in \langle r_i, d_i - p_i \rangle \qquad \forall i \tag{5.20}$$

$$\text{prune}(i, j) \qquad \forall i, j \tag{5.21}$$

Method $\text{prune}(i, j)$ adds either no additional constraint (if task ω_j does not overlap with task ω_i) or just one constraint in order to ensure that the tasks will not overlap. Depending on the properties of the involved tasks, we add a different constraint, see Alg. 1. Finally, if situation σ_3 occurs, there is a flaw in the input set and a schedule is infeasible.

5.2 Control Framework

So far, we have focused only on one component – the scheduler. However, the robot needs other components as well in order to perform tasks. First, it requires navigation controllers in order to autonomously navigate in the environment and to reach tasks locations. Then, the robot needs to have other subsystems in order to able to perform the tasks (e.g. recognise people in the eating area). But mainly, its performance needs to be controlled and monitored. Therefore, we have developed a control framework (Mudrová et al., 2015), which is visualised in Fig. 5.5.

This framework not only controls the execution of the tasks but also it manages the incoming new tasks. Tasks can be added to the control framework by humans via a web interface (e.g. requesting the robot to perform a task at a particular time); by components of the robot's other subsystems (e.g. requesting that a part of the map is explored at some time in the future); and by *routine scripts* which specify fixed sets of tasks for the robot to perform every day. These routine scripts address

Algorithm 1: prune(i, j)

```
1: if $\varepsilon_0 > 0$ then
2:    if $\sigma_0$ and !$\sigma_1$ then                    // $\omega_j$ precedes $\omega_i$
3:       return $s_j + p_j + \mathcal{E}(l_j^e, l_i^s) - s_i \leq 0$
4:    end if
5:    if !$\sigma_0$ and $\sigma_1$ then                    // $\omega_i$ precedes $\omega_j$
6:       return $s_i + p_i + \mathcal{E}(l_i^e, l_j^s) - s_j \leq 0$
7:    end if
8:    if $\sigma_2$ then                                    // both orderings are possible
9:       if $C_0 \leq C_1$ then
10:         return $s_j + p_j + \mathcal{E}(l_j^e, l_i^s) - s_i \leq 0$
11:      else
12:         return $s_i + p_i + \mathcal{E}(l_i^e, l_j^s) - s_j \leq 0$
13:      end if
14:   end if
15:   if $\sigma_3$ then                                    // illegal situation
16:      return infeasible
17:   end if
18: end if
19: return $\emptyset$
```

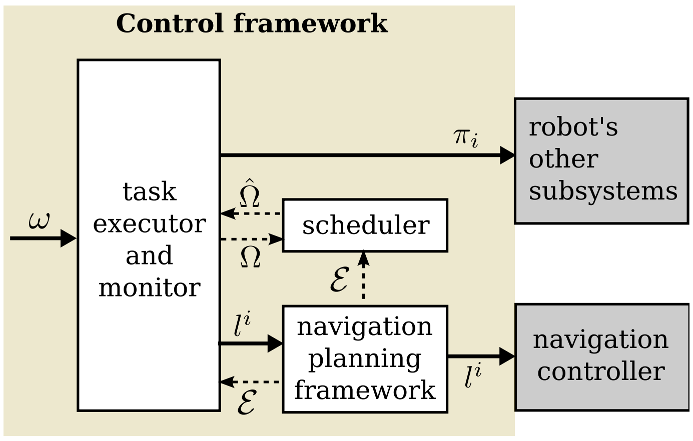


Figure 5.5: The architecture of the proposed control framework, see Sec. 5.2 for an explanation.

periodicity of tasks as illustrated in task ω_2 from the example scenario.

The task executor is the main control component in the framework. It manages task execution and scheduling, whilst also reacting to various forms of failures. When a new task is added to the executor, it updates its schedule and continues with task execution as described by its current schedule $\hat{\Omega}$. The scheduler utilises travel time estimates $\mathcal{E}$ from an optimal topological navigation framework (Lacerda, Parker & Hawes, 2014). This navigation framework uses statistics gathered from long-term experience to adapt $\mathcal{E}$ to the dynamics of the environment.

When the scheduler does not find a solution, we assume it is because the robot is oversubscribed by tasks. Hence, the executor drops tasks with the lowest priorities and tries to call the scheduler again.

Finally, when an on-demand task is added, the task executor cancels the currently executing task (if it is interruptible) and executes the new task instead, whilst rescheduling its remaining tasks, taking the new situation into account.

In the following section, we focus on the aspects of the task executor closely related to handling on-demand and extended tasks. The rest of the components of the control framework are out of the scope of this thesis and we refer the reader to the work in (Mudrová et al., 2015).

5.2.1 Task Executor

The task executor handles the extended tasks via the method *trySchedule()*, see Alg. 2. The method *trySchedule()* calls the scheduler, which tries to find a schedule for a set containing old tasks Ω_o (previously scheduled) and newly added tasks Ω_n. If it does not succeed, method *drop(Ω_n, 0.2)* (Alg. 3), drops 20% of the new tasks with the lowest priority (Ω_d) and overwrites Ω_n with the remaining 80%. The dropped tasks are saved because the robot might still be able to schedule some of them in the future.

Algorithm 2: trySchedule(Ω_n)

```
1: while not scheduler($\{\Omega_o, \Omega_n\}$, $\hat{\Omega}$) and $\Omega_n \neq \emptyset$ do
2:    $\Omega_n$, $\Omega_d$, preemptionNeeded = drop($\Omega_n$, 0.2)
3:    if preemptionNeeded then
4:       $\Omega_n = \Omega_o + \Omega_{sn}$
5:       startExecution()
6:    end if
7: end while
8: return $\hat{\Omega}$
```

The method *drop()* (Alg. 3) operates as follows. First, the method *theLowest-PriorityTasks(Ω_n, amount)* creates a subset Ω_d of tasks with the lowest priority in set Ω_n. If this subset would contain more than $amount \cdot size(\Omega_n)$ tasks, Ω_d is populated by randomly chosen tasks with the lowest priority. Note that Ω_d can also contain fewer tasks than $amount \cdot size(\Omega_n)$ because the method chooses only tasks with the same priority, which is returned as the second parameter. Then, if there is no task currently executing, tasks Ω_d are dropped. However, if there is a task executing, the executor checks whether the tasks to be dropped have a higher priority than the executing task. If they do, then execution must be interrupted so that the higher priority tasks can be propagated to execution. If the executing task cannot be interrupted, the framework is forced to drop the higher priority tasks.

Algorithm 3: drop(Ω_n,amount)

```
1:  $\Omega_d$, priority = theLowestPriorityTasks($\Omega_n$,amount)
2:  if not taskCurrentlyExecuting then
3:      return $\Omega_n \setminus \Omega_d$, $\Omega_d$, false
4:  else
5:      if priority $\leq$ taskExecuting.priority then
6:          return $\Omega_n \setminus \Omega_d$, $\Omega_d$, false
7:      else
8:          if isCurrentTaskInterruptible then
9:              return $\Omega_n$, $\emptyset$, true
10:         else
11:             return $\Omega_n \setminus \Omega_d$, $\Omega_d$, false
12:         end if
13:     end if
14: end if
```

CHAPTER 6

EVALUATION

We compare our proposed scheduler with the proposal of Coltin et al. (2011). In the first part of this evaluation, both schedulers assume that travel times are not varying in time. We evaluate the schedulers using not only simulated data, but also data collected from the long-term deployment of robots in real environments. In the second part, we evaluate how estimates of travel times affect the resulting schedules.

6.1 Standalone Scheduler

We implemented both schedulers (ours and Coltin et al.'s) in C++, using SCIP 3.0.2 (Achterberg, 2009) as a MIP solver. Our code is available as a ROS package[1].

6.1.1 Simulated Data

We have compared the two schedulers on synthetic task sets with the following properties. The input sets of tasks are always feasible. The processing time p_i for task ω_i is generated using uniform distribution between values 2 min and 30 min. The size of the time frame $\langle r_i, d_i \rangle$ is set to be ρ-times bigger than the pro-

[1]https://github.com/strands-project/strands_executive

o	**2**	**10**	**50**	**100**	**200**
proposed $\bar{t}$ [s]	0.044	0.042	0.054	0.133	0.585
Coltin's $\bar{t}$ [s]	0.101	188.116	-	-	-

Table 6.1: Runtime of the schedulers dependent on the size of a task group.

cessing time p_i, when ρ is generated as a random number with uniform distribution from interval $\langle 4, 100 \rangle$. Each set of tasks can be split into multiple groups, where all tasks within a single group have one type of Allen's relation. All tests were run on a Lenovo ThinkPad E-540 with Intel i7402MQ Processor (6MB Cache, 800 MHz).

6.1.1.1. Overlapping Time Windows

The performance of the two algorithms only differs significantly when σ_2 occurs (i.e. when our approach prunes away possible solutions). To explore this case, ten sets containing 200 tasks each were generated. Each set consists of groups of tasks, where the tasks within the group overlap in a way such that the occurrence of situation σ_2 is guaranteed, but tasks from different groups do not overlap. The number of overlapping tasks within the group is denoted by o. The results, presented in Table 6.1, demonstrate that the proposed scheduler is faster than Coltin et al.'s on these problems. Moreover, increasing the group size leads to significantly higher computation times for Coltin et al.'s scheduler. This is mainly due to more combinations of task ordering, which the solver needs to take into account. We have restricted ourselves to a limit of three minutes to provide a solution. Thus, we did not run Coltin et al.'s scheduler for all cases as the time increases exponentially. For the problems that both schedulers solved, the difference in optimality criteria between approaches is negligible.

n	10	20	100	200
proposed $\bar{t}$ [s]	0.003	0.006	0.097	0.585
Coltin's $\bar{t}$ [s]	33.238	196.012	-	-

Table 6.2: Runtime of the schedulers dependent on the amount of equal tasks.

6.1.1.2. Equal Time Windows

The situation when all tasks have similar time frames (i.e. $o = 200$) is even more challenging than the previous situation. This is because there is no difference between time intervals, thus all possible orderings of tasks have the same optimum, but the solver is not aware of this. The results from both schedulers running on sets of equal tasks of increasing size are presented in Table 6.2. A significant time difference between the approaches can be observed even for a set containing only 10 tasks. Again, there is no difference in final optimisation criteria. Because these data were generated to guarantee completeness, both schedulers find solutions on all tested problems.

6.1.2 Data from a Robot

As part of STRANDS project[1], we have deployed two robots in care and security scenarios, where people's health and property are involved (Fig. 6.1). In the care scenario, we cooperated with "Haus der Barmherzigkeit" – a facility for elderly people in Austria (abbreviated HdB). Our robot performed 1985 tasks during 14 days of deployment, including navigating to the chapel or game room, and regularly checking the emergency exits for obstructions. In the security scenario, we

[1]http://strands-project.eu

Figure 6.1: The mobile robots Bob and Henry during their deployment.

cooperated with G4S. In one of their buildings, the robot performed 963 tasks during 16 days. For example, our robot created 3D maps of rooms at scheduled times, checked the position of fire extinguishers, and searched for objects on desks. We gathered all sets of tasks sent to the scheduler during these deployments. Thus, we are now able to compare the two schedulers on these real-world task sets.

Many of the tasks in the sets were generated from a daily *routine* (roughly three-hour slots from 08:45 onwards) given to the robot and thus have large, equal time windows (corresponding to morning, afternoon etc.), see Fig. 6.2. The nature of the routine-based tasks means that situation σ_2 often occurs. A smaller proportion of tasks were generated on-demand by users or other parts of the robot's architecture.

6.1.2.1. Existence of Infeasible Sets

In contrast to the simulated data, infeasible sets can exist in the data from the real environments. The proposed scheduler can only detect infeasible sets via occurrence of situation σ_3, i.e., a pair of tasks cannot be ordered due to their conflicting time constraints. However, more tasks can conflict in such a way that a failure will not be detected while checking pairs, but it will invalidate all the set.

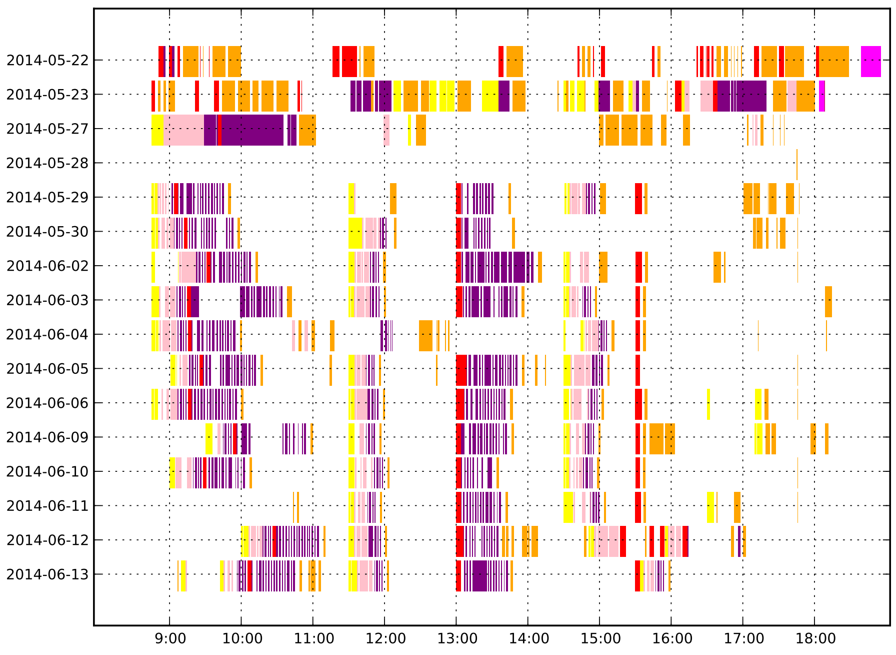


Figure 6.2: The tasks performed by the robot. Key: yellow, object checks; red, door checks; purple, 3D mapping; pink, object search; orange, waiting.

Our scheduler will not be able to detect such a situation beforehand. As a result, an infeasible set can be found only when the underlying solver SCIP returns no solution. This also holds for the Coltin et al.'s scheduler. However, SCIP needs to solve the problem completely, which is limited by the size and nature of problems. Therefore, we cannot determine if any given set is feasible or not. Instead, we compare if the schedulers fail to find a solution within the three-minute limit. Table 6.3 presents counts of three possible outcomes:

- both schedulers fail;
- the proposed scheduler fails but Coltin et al.'s succeeds, (which occurs mainly for sets containing a small amount of tasks);
- Coltin et al.'s scheduler fails but the one proposed succeeds (which occurs mainly for sets containing a large amount of tasks).

Dataset	Number of Sets	Both failed	Proposed failed	Coltin's failed
HdB	606	103	24	33
G4S	358	14	2	3

Table 6.3: Comparison of cases where no valid schedule is found

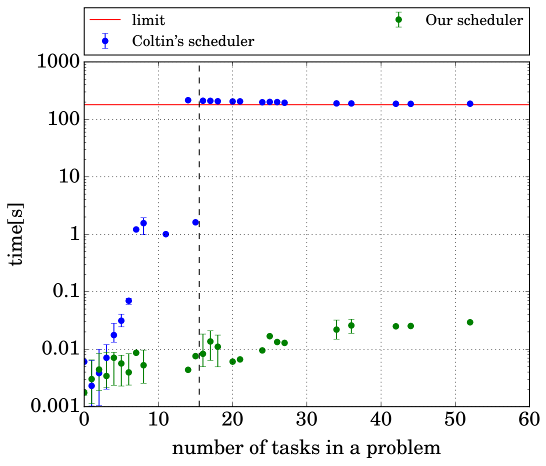
However, there cannot be made any conclusion about their behaviour as the data have not shown any significant difference.

6.1.2.2. Speed of the Schedulers

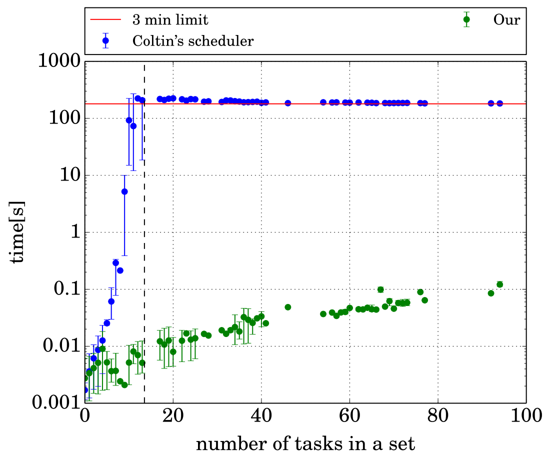
We ran both schedulers on the collected data and we recorded their runtime. Because the collected data includes problems with the same number of tasks, we report on the average performance and the corresponding variances. Moreover, when a scheduler reached the three-minute time limit, we stopped it. However, the underlying SCIP solver was still able to provide a solution in such cases, but it cannot determine if this is the optimal one or not. This is the case of Coltin et al.'s scheduler on many problems, see Fig. 6.3a and 6.3b (for the G4S and HdB data, respectively).

On the one hand, the problems to the left of the black dashed line are solved *completely* by the Coltin et al.'s scheduler. The word "completely" refers to the fact that the SCIP solver returns the optimum criterion for the specified MIP (which itself may be non-optimal for the problem). On the other hand, the scheduler solving the problems to the right of the black dashed line has been stopped due to the runtime limit.

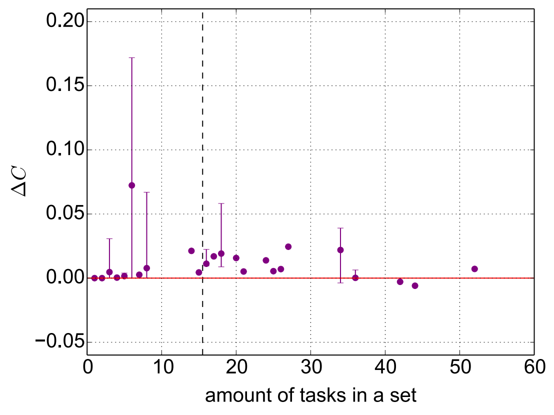
We can conclude that our proposed algorithm is significantly faster than Coltin et al.'s as the size input set increases. Moreover, the MIP problem specified by the proposed scheduler is *completely* solved within the three-minute time limit for all sets. Furthermore, the variances of runtime on the data are small. This illustrates how the proposed scheduler has deterministic behaviour. We have discussed in the introduction that such property is beneficial when batches are scheduled (in order to estimate how long a scheduler needs to provide a solution for a given problem).



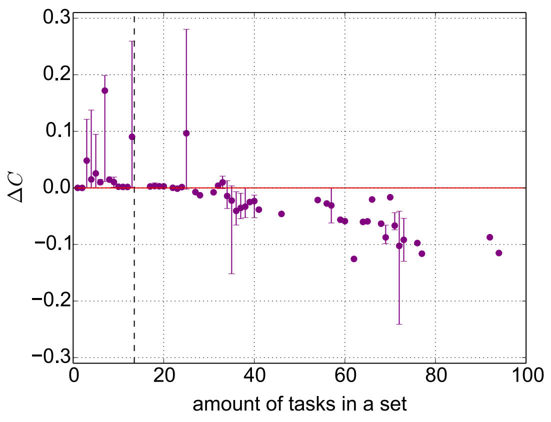
(a) G4S - runtime



(b) HdB - runtime



(c) G4S - criterion



(d) HdB - criterion

Figure 6.3: The comparison of the schedulers in the G4S and HdB scenarios based on their speed and quality of the found schedules.

6.1.2.3. Quality of the schedules

Even though we have stated that we are interested in a fast scheduler rather than optimal one, the scheduler still needs to perform reasonably well. We can determine which scheduler performs better on a task set by comparing the optimisation criteria of the resulting schedules for such sets when both algorithms have found a schedule. We performed a Wilcoxon signed rank paired test to analyse if there is a significant difference between the criteria found by our and Coltin's et al.'s scheduler. For the G4S set, we can conclude that there is a significant difference because $p-$value returned by the test is $p = 1.85 \cdot 10^{-16}$ whilst $T = 407$. Our scheduler performs worse; the mean value of the differences between our scheduler and Coltin et al.'s is $\mu = 0.32$ h (hours) and the standard deviation is $\sigma = 0.83$ h. In contrast, we can conclude the exact opposite about the results on the HdB test, where our scheduler is significantly better than Coltin's et al.'s as the Wilcoxon test returns p-value of $8.66 \cdot 10^{-14}$ and $T = 13227$. The mean value of the differences is $\mu = -0.80$ h and the standard deviation is $\sigma = 2.23$ h. The different outcome is affected by the nature of the input data. Whilst tasks' time windows significantly overlap (or are even equal) in the G4S set, most of the tasks' time windows do not overlap in the HdB set.

In order to visualise the differences between criteria, we compute the following normalised metric because the absolute value of this criteria varies by task set.

$$\Delta C = \frac{C_p - C_c}{C_h}, \tag{6.1}$$

where:

- C_p is the optimisation criterion of the result from the proposed scheduler,
- C_c is the optimisation criterion of the result from Coltin et al.'s scheduler,
- C_h is the highest (i.e. worst) possible optimisation criterion for the input set Ω, which is computed as:

  $$C_h = \sum_{i \in \{1 \dots n\}} (d_i)$$

  for n tasks in set Ω.

The property $\Delta C \in \langle -1.0, 1.0 \rangle$ holds. This metric can be computed only for such sets when both algorithms have found a schedule. Negative values correspond to the fact that the proposed algorithm has found a better criterion than Coltin et al.'s and vice versa.

The ΔC values are displayed in lower graphs in Figures 6.3c and 6.3d. In the graphs, sets to the left of the black dashed line are those with small amounts of tasks for which the solver has found the *optimal* solution for both schedulers. Sets to the right of the line are those for which the solver has found *some* solution for Coltin's approach and the *optimal* one for the proposed problem.

6.1.3 Summary

We have demonstrated that the proposed scheduler performs significantly faster than Coltin's et al.'s not only on the simulated data, but also on the data collected during deployment of two robots in the real world scenarios. A conclusion about the quality of the found schedules cannot be made as more data are needed in order to study influence of tasks' properties to the resulting criteria.

6.2 Scheduler with Travel Times Estimates

In this section, we evaluate the impact of the estimates of travel times on the schedules. Recall that the travel times estimates are time dependent as the environment dynamics affect the robot navigation. Therefore, we use only data gathered in the security scenario, as this environment is smaller than HdB and the robot is affected more by the dynamics. To summarise, the security environment has approximately 300 m^2 and is used by approximately 20 people. The topological map, on which the robot travelled, is in Fig. 6.4. During the three-week deployment, the robot covered 20.64km and completed 963 tasks successfully.

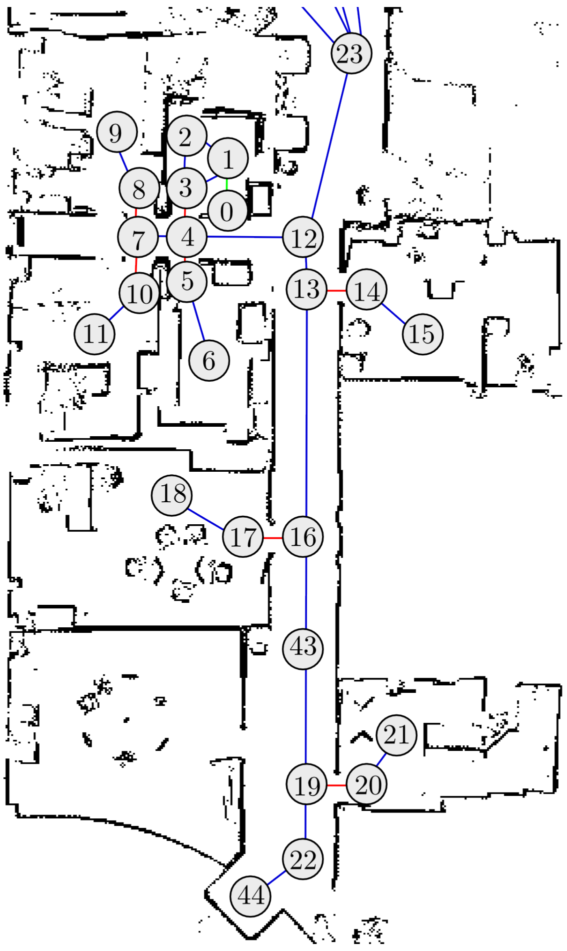


Figure 6.4: A detail of the topological map traversed by the robot in the security scenario. Blue edges are within an open space, but the red edges highlight doors.

6.2.1 Influence of Environment Dynamics

The navigation planning framework learns travel times from the previous experiences. Therefore, the expectation is that the framework will provide better estimates as it collects more data throughout the deployment. To evaluate this hypothesis, we compare real travel times collected during the third week of deployment with three estimates provided by the framework, which uses only

- distance-based estimates, no data have been learned ($W0$);
- the first week to learn ($W1$);
- two weeks to learn ($W2$).

We evaluate the performance on 435 paths that the robot travelled during the testing week. For each path, we obtain the ground truth value g from the dataset which states how long the navigation actually took. Then, a set of time estimates $\mathcal{E}_X$ on each path is obtained from the navigation planning framework, where $X = \{W0, W1, W2\}$.

6.2.1.1. Methodology

The relative difference between an estimate and the ground truth is calculated as follows:

$$\Delta\mathcal{E}_X = \frac{\mathcal{E}_X - g}{g}. \tag{6.2}$$

Positive values of $\Delta\mathcal{E}_X$ mean that the particular method overestimates the time needed to travel between tasks. In terms of robot behaviour this typically means the robot will arrive early for tasks and must, therefore, wait around (wasting time) before executing the associated actions. In contrast, negative values mean that a schedule is created that underestimates the time needed to travel between task locations. As a result, the robot will fail to successfully complete the schedule.

set	0.1-q	0.25-q	0.75-q	0.9-q	error	std-dev
$\Delta\mathcal{E}_{W0}$	-0.457	0.141	1.844	2.714	1.155	0.895
$\Delta\mathcal{E}_{W1}$	-0.151	0.244	2.088	12.823	8.522	30.526
$\Delta\mathcal{E}_{W2}$	-0.189	0.227	1.171	1.896	0.982	1.082

Table 6.4: The results of travel time estimates on the dataset.

Therefore underestimates are a more damaging form of estimation error in our framework.

6.2.1.2. Results

The absolute mean errors and corresponding standard deviations of the relative results, plus their quantiles, on the 435 paths are reported in Tab. 6.4. Estimates for selected paths from Fig. 6.4 are visualised in Fig. 6.5. It can be observed that the planning framework with the most training data, $\Delta\mathcal{E}_{W2}$, provides the best estimates (lowest mean error). As would be expected, the non-adaptive model based on distance $\Delta\mathcal{E}_{W0}$ underestimates by a larger degree to adaptive model using W2.

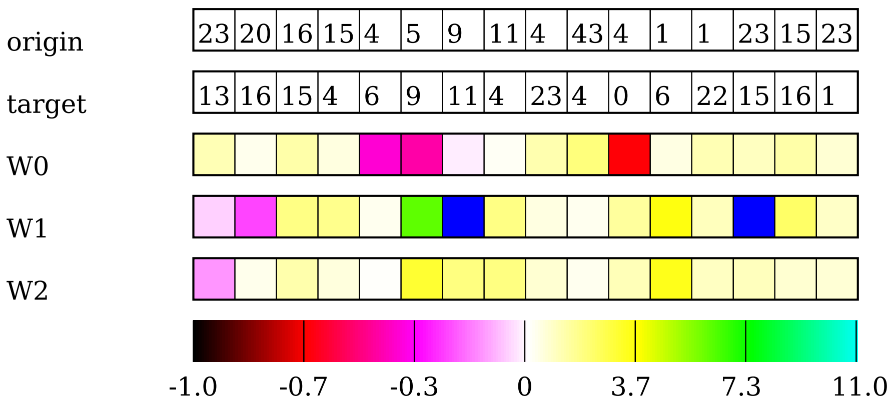


Figure 6.5: Selected estimated relative travel times $\Delta\mathcal{E}_X$. For example, the travel time for the path from node 4 to 0 is underestimated in the data $W0$, as the nodes are relatively close to each other. However, the robot had difficulty to navigate between them due to very restricted space. However, this estimate is adjusted in the $W1$ and $W2$ estimates.

After one week of data the adaptive model ($\Delta\mathcal{E}_{W1}$) substantially overestimates most of the paths. When the second week of data is added, more experience of environment causes the estimates to increase in accuracy. $\Delta\mathcal{E}_{W2}$ still overestimates but significantly less than model $\Delta\mathcal{E}_{W1}$. As can be observed in Fig. 6.2, during the first days of the deployment the routine was not followed very well (due to bugs in other parts of the system). Due to this, the data available for training $\Delta\mathcal{E}_{W1}$ has a different temporal distribution to the testing data. The second week of data corrects this, demonstrating that *long-term* experience improves the approach considered in the planning framework.

6.2.2 Adaptive Scheduling using Travel Time Estimations

Finally, we are able to illustrate how these travel time estimates affect schedules. To illustrate this, we generated a test scheduling problem, comprised of 16 tasks across 13 different locations. Additionally, we schedule those tasks in three different time slots: morning, after lunch and late afternoon. Our assumption is that dynamics of the environment is different in these slots, hence the schedules will be different too as a result of learned time estimates considering these dynamics.

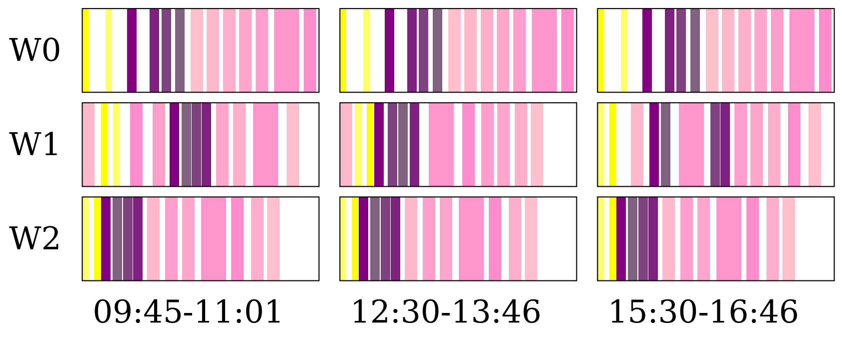


Figure 6.6: Schedules using different methods for travel time estimates. White space between tasks represent the expected time to travel between tasks' locations. Yellow represents object checks; purple 3D mapping; pink object search. Finally, different shades of tasks represents that tasks were in different locations.

The schedules computed for different times of day are depicted in Fig. 6.6. It can be observed that the set of tasks can be scheduled more compactly as the expected travel times get overestimated by less as a result of the larger amount of data used for learning. Furthermore, the results with one week of learning data illustrate the adaptive nature of our framework: the schedules are different throughout the day. However, this variation after one week of learning is largely incorrect. Because our environment was not very dynamic after two weeks of data the schedules look the same throughout the day. Notice though that they are different to the schedules based on distance estimates, as our model adapts to the robot's behaviour in the environment.

6.2.3 Summary

In this section, we have illustrated how the proposed scheduler may benefit from a cooperation with the navigation planning framework, which learns about the environment dynamics and adapts its estimates of navigation times.

CHAPTER 7

CONCLUSION AND DISCUSSION

In this part of the thesis, we presented a novel fast scheduler for use on mobile robots. The main contribution of our work is the use of Allen's interval algebra to prune possible solutions within the scheduler. However, this heuristic approach does not guarantee to find an optimal solution and it may loose completeness, which can be recovered by using the found solution to warm start full MIP (used in Coltin et al.'s scheduler). Nevertheless, we prefer faster heuristic approach rather than an optimal approach as finding the optimal solution may be considerably more time consuming, especially for larger problems.

Note that our pairwise task analysis results in a total-order on tasks. Hence, starting times to each task could be assigned via an iterative approach rather than LP. The iterative approach means that the starting time s_j of task ω_j can be established as the maximal value of (i) either the sum of the ending time of its preceding task ω_i and the travel time required between them; (ii) or its release date r_j. However, using LP allows us to use only some features of our proposed heuristic. For example, the pruning of situation σ_2 can be omitted, especially in the domains where this step leads to many unsolved problems.

Our experimental results have shown that the proposed scheduler is able to quickly solve task sets with large numbers of tasks, which significantly outperforms the state-of-the-art scheduler designed for the same domain. The proposed scheduler is open source and was integrated into several mobile robots deployed in the real world scenarios. In particular, we reported on the deployment in two scenarios where two robots successfully performed 2948 tasks within 30 days. Finally, we have illustrated how the scheduler benefits from learned travel estimates and how this results in varying schedules, depending on the time of the day.

7.1 Limitations and Future Work

The introduced assumptions A1 - A4, A8 and A9 are the limitations of our work. Our problem formulated via MIP can be extended in order to handle the cooperation between several robots (A1) as illustrated in (Korsah et al., 2012). Davenport and Beck (2000) surveyed scheduling techniques for handling uncertainty (A2), whilst the survey of (Laborie, 2003) focused on allocating resources (A3). The inseparability of some tasks (A4) can be added into our MIP problem as an additional constraint. If tasks were not added in batches (A8) and affected by uncertainty, we will first need to extract the tasks which are certain. We could use techniques of *least commitment and delayed commitment scheduling* (Berry, 1993). Finally, the atomicity of tasks (A9) can be overcome by exploiting temporal planning techniques. A deliberation system building on temporal planning is developed in the next part of our thesis.

Part II

Temporal Planning

CHAPTER 8

INTRODUCTION

In this part of the thesis, tasks are no longer atomic actions, but they are defined by *goals*. A goal represents a certain state of the robot or its environment that needs to be achieved. For example, the goal of task ω_4 from the example scenario is that John has the cereal bar before 14:00. As a result, our system (to be developed and evaluated in this part of the thesis) takes control over creating *plans* achieving tasks' goals. The main advantage of this system, in contrast to the proposed scheduler, is that it can reason over the tasks' plans. As a result, it recognises and exploits similarities between different tasks. As we will demonstrate, such an ability can significantly enhance the robot's performance.

In order to decide *how* and *when* tasks are going to be performed, our system builds on *temporal planning*. Temporal planning can be addressed either via *state-based* or *time-based* approaches. We have analysed both of them and we report on the findings in the next chapter, where we also justify why we have decided to focus on the state-based approach.

In contrast to scheduling, a support of time windows in state-based temporal planning is limited. This can be illustrated on the *cooking* domain from the International Planning Competition (IPC 2011)[1]. This domain describes a problem where a meal must be ready before 20:00 but no sooner than 19:55, otherwise the meal

[1]http://www.plg.inf.uc3m.es/ipc2011-deterministic/

will be cold. However, the constraints are imposed only on a meal being ready. This is in contrast to our example scenario, where we require that the whole plan is restricted by the time window, because release dates represent some external reason why a task cannot start sooner. Recall that this property is reflected in the assumption **A5** and must be obeyed by our system.

In comparison to scheduling, temporal planning significantly increases the difficulty of the addressed problem, because it substantially adds to the size of the search space. Another aspect increasing the difficulty is that actions can overlap in time in a temporal plan. The literature differentiates between two terms:

- two actions are in *parallel* if (i) they overlap in time and (ii) there are no causal or temporal constraints between them;
- two actions are *required to be concurrent* if (i) they overlap in time and (ii) a causal or temporal constraint requires a specific interaction between the actions.

We refer to a *sequential* plan if there are no parallel or concurrent actions in the plan. *Required concurrency* can be illustrated on the example from the *turn and open* domain from the IPC 2014[1], where a robot with a gripper is required to open a door. In order to do so, it needs to turn the door knob and hold it turned whilst it pushes the door. Hence, the action push is required to be ordered *during* action turn. The synchronisation of actions is critical for the success.

Required concurrency and time windows limiting a whole task are respected by our proposed merging algorithm, which operates as follows. First, it obtains plans for each task separately by utilising an off-the-shelf temporal planner. Hence, the planner operates on small search spaces which contributes to good scalability. Then, our algorithm merges these plans into a final plan while respecting their causal and temporal constraints. This is done by *reasoning* over the input

[1]https://helios.hud.ac.uk/scommv/IPC-14/

plans and their related time windows, and finding *synergies* and *demands* between different tasks.

8.1 Motivation Example

Consider a robot that receives two tasks ω_1: "notify people in a common eating area to leave between 20:00 - 20:15" and ω_2: "check if the emergency exits are clear every 2 hours" from the example scenario (Sec. 1.1). The periodic property of task ω_2 is handled in the executor, as discussed in Sec. 5.2. This executor adds the appropriate task into the system every two hours, hence we assume an instance of ω_2, which is required to happen between 20:00 - 22:00. The problem of finding a single temporal plan satisfying these tasks can be solved by a range of approaches that are going to be illustrated here.

1. A *sequential* approach, similar to scheduling in Part I, could be reapplied here. Assuming domains which do not consider time windows of tasks, the order of tasks can be resolved by using a simpler algorithm, such as first-in-first-out (FIFO) approach, rather than scheduling. Then, exploiting assumption **A4** (tasks are separable), a task planner would plan for each task separately, considering the final state reached by the previous plan. Such a situation is illustrated in Fig. 8.1a where the plan for task ω_1 precedes ω_2. The makespan of the sequence is 9.5 min. This approach scales well due to the combination of small search spaces. However, it obtains a bigger makespan than necessary. This approach is common for mobile service robots, e.g., (M. M. Veloso et al., 2012), due to its simplicity and efficiency.

2. This scenario could be addressed by a temporal planner. First, all task goals are *conjoined* into a single planning problem. Then, a temporal planner provides a single plan, which satisfies all task goals. The disadvantage of this

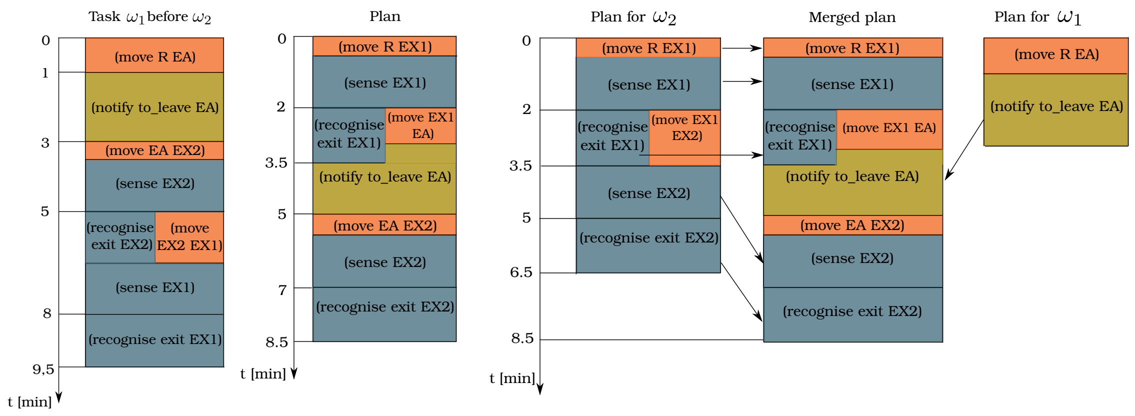
approach is the size of the search space which leads to a poorer scalability in comparison to the sequential approach. In contrast, the main advantage is that it does not require assumption **A4**, i.e., it can handle inseparable tasks as well. Moreover, this approach can find an optimal solution, i.e., the best makespan of the final plan, when an optimal planner is used.

The makespan of the plan for tasks ω_1 and ω_2 is 8.5 min (Fig. 8.1b). The saving of 1 min, in comparison to the sequencing approach, is obtained because actions related to different tasks can be interleaved. Hence, the robot first checks the exit in location EX1 and, on its way to location EX2, it stops by location EA to satisfy task ω_1.

3. Our proposed merging approach finds high quality of plans similarly to the conjoining approach (Fig. 8.1c). The approach has two important features: (i) it does not have to merge all actions from the input plans and (ii) it can add some new actions in order to satisfy causality of the merged plan. We elaborate on these features in Chapter 12. In this example, action (move EX1 EX2) from ω_2 and action (move R EA) from ω_1 has not been merged. In contrast, action (move EX1 EA) has been added in order to link task ω_1 into the plan of ω_2.

8.2 Research Scope

The merging approach is not novel; for example, Koehler and Hoffmann (2000) discuss that considering interactions between goals can decrease the complexity of finding the solution plan. However, they also prove that finding an ideal ordering between goals is as difficult as finding a plan. Therefore, our algorithm is driven by a proposed heuristic. In this thesis, we build on the insights into plan merging and we answer the following research questions:



(a) Sequencing (b) Conjoining (c) Merging

Figure 8.1: The comparison of sequencing, conjoining and merging approaches how to obtain a plan for tasks ω_1 and ω_2. The arrows in (c) represent what actions from the input plans have been reused in the merged plan.

Q1 Given plans for each task, how can these plans be *merged* into a single final plan satisfying all task goals and time constraints?

Q2 Is the scalability of the merging approach better or worse than scalability of *sequencing* and *conjoining* approaches?

Q3 What is the quality of the final merged plan when comparing to the plans found by *sequencing* and *conjoining* approaches?

Q4 How do all the approaches handle time windows?

8.3 Organisation

This part of the thesis is organised as follows. Chapter 9 analyses the related work. First, it provides a comparison of state-based and time-based planning approaches. This is important as our algorithm wraps a state-of-the-art planner to provide it with plans for the input tasks. Hence, the choice of the representation and the planner strongly affects the developed algorithm. Then, existing techniques of plan merging are reviewed. The chapter is concluded by a discussion of why we build on state-based temporal planning. The nomenclature of state-based planning is formally defined in Chapter 10. It is followed by formal definition of the addressed problem and formalisation of the three solution approaches (Chapter 11). The proposed merging algorithm is described in Chapter. 12 and evaluated on existing and novel planning problems (Chapter 13). We conclude by the discussion of advantages and disadvantages of the proposed algorithm (Chapter 14).

CHAPTER 9

RELATED WORK

In our endeavour to choose a representation and a planner to be integrated with our merging algorithm, we compare the state-based and time-based approaches. We analyse their theoretical properties, modelling languages and corresponding planners. We assume that the reader is familiar with classical planning (non-temporal). If not, we refer the reader to the book by Nau, Ghallab and Traverso (2004). Classical planning is going to be referred as *instantaneous planning* for the rest of this thesis (in order to highlight that it does not address time). Moreover, we analyse existing merging techniques for both representations. We conclude this chapter by choosing a representation to be used for the rest of this thesis.

9.1 Temporal Planning

The two possible views of temporal planning are state-based and time-based. The difference between these two approaches is visualised in Fig. 9.1. The state-based approach focuses on states; hence, it projects into the vertical axis. It samples time in order to make decisions on the static states of the world. In contrast, the time-based approach focuses on the time axis. In this approach, a state

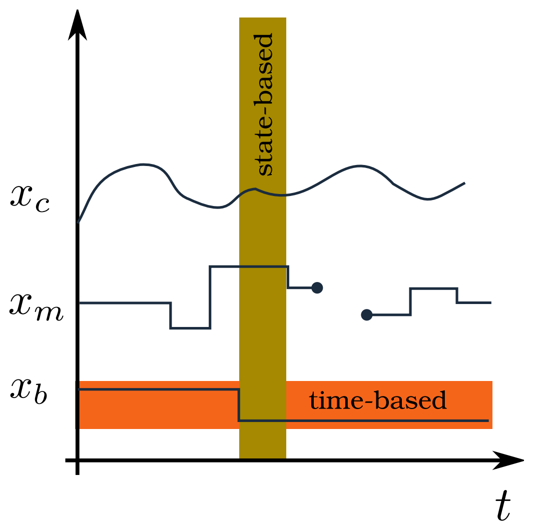


Figure 9.1: The state-based approach to temporal planning operates only with a sampled view of a state variable x. In contrast, the time-based approach views the state variable as a temporal function.

variable is described by a time-varying piecewise function, i.e., gaps may exist in the state-variable specification where the variable is unspecified.

In our comparison, we are going to refer to a number of properties how planners can be classified:

- A planner is either *offline* or *online* depending on, whether planning is fully finished before the execution of the plan starts.

- Closely related to offline and online planning is if a planner plans for several tasks at once, which is denoted as a *batch* planning, or if it has a *continuous* approach, where it plans one task by another. Thus, continuous planning can be exploited in online planning and batch planning is often done offline.

- A planner provides either a single plan or multiple plans, i.e., it keeps improving the quality of the found plan. The latter case is referred to as *anytime* planning. An anytime planner will terminate only if it searches all its search space. Hence, it can run for very long time. Therefore, these planners are normally limited in their runtime and memory usage in order to stop them sooner before all search space is searched.

9.1.1 State-Based Approach: PDDL-based planners

State-based temporal planning extends the instantaneous planning by four aspects:

1. It removes the assumption that actions are instantaneous; instead they last for a certain amount of time, which is called *duration* of an action. This raises the main complication in temporal planning. A temporal planner needs to decide not only *what* action to perform but also *when*. Moreover, actions may be ordered in parallel or in *required concurrency*.

2. Preconditions and effects of a durative action are associated with a time instant. For example, effects of an action are applied at the start of the action.

3. A temporal planner assigns a time stamp to each action representing when the action can start. Hence, such a plan is *scheduled*. Moreover, a scheduled plan needs to obey imposed time constraints, such as *release dates* and *deadlines*.

4. The state variables are generally time-dependent, hence they are *continuous* functions. However, the literature strictly differentiates between *temporal planning* and *hybrid planning*. Temporal planning assumes that state variables are only multi-valued piecewise functions and that their change is caused only by effects of actions. In contrast, hybrid planning addresses state variables that are time-dependent continuous functions (Fox & Long, 2006). Hybrid planning is out of the scope of this thesis.

While all modelling languages for temporal planning support the first aspect, they strongly differ in modelling the second. Therefore, W. Cushing, Kambhampati, Mausam and Weld (2007) propose classification of temporal modelling languages to either temporally simple or temporally expressive. A language is temporally expressive if it supports modelling of an action where there is no single time point during the action's duration when all preconditions and effects hold together. That means that there is a *temporal gap* between time instances when preconditions or effects hold. In contrast, a modelling language is *temporally simple* if it does not support temporal gaps.

9.1.1.1. Planning Domain Definition Language

Despite the existence of different modelling languages for state-based temporal planning, such as TGP (Smith and Weld, 1999) or SAS$^+$ (Bäckström and Nebel, 1995), we focus on the Planning Domain Definition Language (PDDL). We choose PDDL as it is a temporally expressive language. Moreover, many PDDL-based planners exist and their binaries are available. Hence, we can easily compare them to our proposed algorithm. The popularity of temporal PDDL-based planners lies in the exploitation of the existing search techniques from instantaneous planning and their domain independent heuristics.

Time has been introduced to PDDL in version 2.1. level three (Fox & Long, 2003) in order to model durative actions with temporal gaps. Preconditions and effects are assigned to a start and an end instances of an action, but not to any arbitrary time stamp during the action. This is seen as a limitation by Smith (2016). Smith argues that preconditions and effects must take place at any arbitrary moment rather than just at the start or the end of an action (in order to support modelling of the real world). However, this drawback can be overcome by splitting one durative action into more. Furthermore, PDDL 2.2. (Edelkamp & Hoffmann, 2004) introduced *timed initial literals* in order to constrain a literal in time. This feature can be used to model release dates and deadlines. Finally, PDDL 3.0 (A. Gerevini & Long, 2005) introduced *preferences* on literals that can be used to model deadlines.

9.1.1.2. Classification of Temporal Planners

Most of the state-based temporal planners are forward-chaining, as this approach has proven to be very successful in instantaneous planning. Additionally, forward-chaining temporal planners can be further classified into three categories based on how they handle relations between actions (W. A. Cushing, 2012): first fit planners, decision epoch planners and temporally-lifted planners. However, other concepts, such as search in plan-space or planning via satisfiability, have found their way into temporal planning as well.

9.1.1.3. Forward-Chaining: First Fit Planners

When solving a temporal planning problem, first fit planners ignore the durations of actions and obtain an instantaneous plan. The temporal plan is obtained by stretching the instantaneous plan to correspond with durations of the actions. The main benefit of this approach is its speed as it can reuse very successful instantaneous planners. In contrast, the main disadvantage is its inability to cope with required concurrency, which cannot be handled because instantaneous plans are *sequential* by nature. Recall that required concurrency demands that specific actions must be ordered concurrently. Some first fit planners at least reschedule the solution plan. Hence, some actions can be ordered in parallel. Nevertheless, rescheduling is still unable to handle required concurrency.

In spite of this strong simplification, such planners are winning the temporal track of the International Planning Competition (IPC) since 2006, where SGPlan (Hsu, Wah, Huang & Chen, 2006) has won. It was followed by an unofficial planner based on MetricFF in 2008 (Hoffmann, 2002) and YAHSP (Vidal, 2014).

All of these planners use additional insights of how to improve their performances. SGPlan first splits a planning problem by parallel decomposition into loosely joined subproblems. Then, it solves these subproblems by using a variant of the Metric-FF. The YAHSP (Yet Another Heuristic Search Planner) (Vidal, 2004) extends the very popular FF algorithm (Hoffmann & Nebel, 2001). FF exploits a *relaxed planning graph* (RPG) constructed by ignoring the delete effects of actions. While FF planner uses only some actions from this graph, YAHSP reuses as many actions as possible in order to keep the search complete.

9.1.1.4. Forward-chaining: Decision Epoch Planners

Two temporal actions can have thirteen possible relations (Allen, 1983). Decision epoch planners (DEP) considers only a subset of these relations. Therefore, these planners are unable to handle required concurrency. In a forward-chaining search, they assume that an action can start either at the same time instance as another one starts or immediately after another one ends. In a backward approach, an action can end either at the same time instance as another one ends or immediately before another one starts.

Decision epoch planning has been introduced in TLPlan (Bacchus and Ady, 2001) and followed by many popular planners such as SAPA (Do and Kambhampati, 2011), LPGP (Long and Fox, 2003), which picks decision-epochs forward and backwards through time, and TFD (Eyerich et al., 2009). Even though TFD (Temporal Fast Downward) (Eyerich et al., 2009) utilises a SAS$^+$ representation , we include it in our comparison as it is PDDL compatible, i.e., PDDL can be automatically translated into SAS$^+$.

9.1.1.5. Forward-chaining: Temporally-Lifted Planners

Temporally-lifted planners support all relations between actions. Hence, they support required concurrency. These planners handle actions as their start and end instances. The temporal relations between the actions instances are preserved in a Simple Temporal Network (STN) (Dechter et al., 1991). This network models not only dependency between instances of the same action, but also of different actions. Therefore, the consistent STN provides the necessary ordering of actions in a solution plan.

Crikey (A. Coles, Fox, Long and Smith, 2008) is the first planner, which has introduced this approach, and has been followed by a family of planners inheriting from it, such as POPF (A. J. Coles et al., 2010) and OPTIC (Benton et al., 2012). Furthermore, these planners combine advantages of partial order planning (POP) with forward chaining techniques. Hence, they find a partial order plan.

The OPTIC planner (Optimizing Preferences and TIme-dependent Costs) (Benton et al., 2012) extends POPF by supporting preferences defined in PDDL 3.0 and a continuous penalty. Therefore, OPTIC utilises Mixed Integer Programming (MIP) solvers. Currently, there can be two versions of OPTIC based on the used solver, either open-source CLP or IBM CPLEX. They significantly affect OPTIC performance. Even though, OPTIC with CLP should be generally faster (according to the authors[1]), we use OPTIC with CPLEX as it is needed in order to support PDDL 2.2 and 3.1.

The POPF planner (A. J. Coles et al., 2010) combines the advantages of forward-chaining planners with least-commitment of partial order planning. Reduction of the commitment during forward search is achieved by adding only temporal constraints that ensure that the preconditions of an action are met. It employs a Temporal Relaxed Planning Graph (TRPG) (A. I. Coles, Fox, Long and Smith, 2008) as its heuristics.

9.1.1.6. Planning as Satisfiability

The ITSAT planner (Rankooh and Ghassem-Sani, 2015) addresses the temporal planning problem with required concurrency. It uses the temporally-lifted approach, i.e., durative actions are split into the corresponding instances and the

[1]https://nms.kcl.ac.uk/planning/software/optic.html

appropriate STN is built. This simplified problem is then encoded into a SAT formula (Rankooh and Ghassem-Sani, 2015). The algorithm keeps iterating until a temporally valid plan is found.

9.1.1.7. Plan-space Planning

The VHPOP (Versatile Heuristic Partial Order Planner) (Simmons and Younes, 2011) builds on partial order planning (POP). The biggest disadvantage of general POP is a lack of heuristics on how to choose a plan to expand or a flaw within the plan to fix. This is overcome by the VHPOP, by exploiting the A* algorithm with several heuristics in order to make these choices in an informed way. This leads to a significant reduction of planning effort. The temporal constraints are saved in an STN as in the case of OPTIC and ITSAT.

9.1.1.8. Planner Expansions

Some successful planning systems wrap an off-the-shelf planner rather than being a planner per se. Their focus is on pre-processing the planning problem in order to make the integrated planner more efficient. Such an approach is taken by DAE$_X$ (Divide and Evolve) framework (Bibaï, Savéant, Schoenauer and Vidal, 2010), which encloses planner X to solve subproblems that are created by an evolutionary algorithm. If the integrated planner fails to find a solution to a subproblem, an evolutionary algorithm makes a new generation of subproblems. There is no guarantee that generated subproblems will be easier than the original one. DAE$_X$ adopts the capability to handle concurrent actions from the enclosed planner.

9.1.2 Time-Based Approach: Timelines planners

Timeline planning has been introduced in the HSTS (Heuristic Scheduling Testbed System) (Muscettola, 1993). It is inspired by control theory, where a system is modelled using multi-valued state variables to describe its dynamic properties. A timeline plan finds a set of relevant state variables and their temporal developments. Moreover, the plan needs to obey given *synchronisation domain rules* expressing temporal and causal constraints. For example, in order to make a cup of coffee, a causal constraint is that a robot needs to have a mug before pouring hot water out of a kettle.

Timeline representation can be summarised as follows (Cialdea Mayer, Orlandini & Umbrico, 2016). State variables are described as piecewise functions varying over time. A variable is defined by its value over an interval, denoted as a *token*, and its possible values are limited to a finite discrete domain. An ordered set of tokens for a state variable is called a *timeline* when the order is established by causal and temporal constraints between tokens. Maintaining the conflict-free constraints between tokens, i.e. *synchronising* them, is how a timeline planner obtains a plan. Importantly, a timeline planner provides a *set* of valid plans rather than a single plan as a state-based planner. The set of plans is denoted as a *chronicle* (Nau et al., 2004).

A different approach how to control a discrete variable over time is taken by the IxTeT planner (Malik Ghallab & Laruelle, 1994). IxTeT and HSTS are the foundation stones for modern timelines planning. IxTeT tackles timeline planning via *temporal assertions* on multi-valued state variables. A variable can either *hold* the same value during an interval or it can instantaneously change via occurrence of an *event*. Both assertions are expressed as predicates followed by terms in order to specify a variable, its values and time instances as the IxTeT builds on a point-based time logic, rather than interval logic as tokens. Moreover, IxTeT is closer to a planning community and inherits model of *operators* (or so called actions in PDDL). Similarly to PDDL, an operator has preconditions and effects. Operators have the same functionality as synchronisation rules. They impose different causal and temporal constraints on the solution plan.

9.1.2.1. Timelines Planners

Dependencies between tokens in a timeline or between assertions can be naturally captured by constraint networks. A constraint network can model not only the requirement for a token to be within a specific time window but also the synchronisation rules between tokens. However, tokens are not limited only by their temporal constraints. Some planners (such as IxTeT) also consider binding constraints between different variables. Moreover, novel planners add other constraints as well, such as spatial and resource. Hence, a mixture of different types of constraints raises an interesting issue: should a planner treat different types collectively or separately?

The main benefit of treating constraints separately is that existing algorithms can be used to handle the constraints. For example, Simple Temporal Networks (STNs) (Dechter et al., 1991) are very efficient in solving temporal constraints. However, separate networks need to be synchronised. To overcome this issue, S. Fratini, Pecora and Cesta (2008) propose to model different variables as a single abstract *component*. We classify existing planners based on the fact if they utilise multiple or a single network.

Timeline planners search in the space of constraint networks. A search state is a constraint network (or more synchronised networks). It is expanded either by adding a constraint to fulfil synchronisation rules or by adding a new token. The fact that a search node is a constraint network has an advantage that the search space is more compact, in comparison to the state-based planners, as a single search node represents many partial plans. Hence, a leaf search node represents a set of possible plans, rather than a single one. In contrast to a state-based planner, a timeline planner searches for *feasible* plans rather than the best plan. This may be a disadvantage in some applications, where a quality of the solution plan matters.

9.1.2.2. Planners with Multiple Constraint Networks

The IxTeT planner uses two separate CSP networks: temporal and binding. Its representation is extended by HTN planning within the FAPE (Flexible Acting and Planning Environment) (Dvořák, Bit-Monnot, Ingrand & Ghallab, 2014). It is the only planner that implements the ANML language (Action Notation Modeling Language) (Smith, Frank & Cushing, 2008)). Additionally, the FAPE also focuses on temporal uncertainty and execution (Bit-Monnot, 2016).

Whilst IxTeT and the FAPE are both academic planners, timelines planners are very popular by space agencies to plan for complicated real world missions. The HSTS system demonstrates its importance on an example of planning of observations for the Hubble Telescope. The NASA has extended the HSTS into the EUROPA (Extensible Universal Remote Operations Planning Architecture) (Barreiro et al., 2012), which is a framework publicly available that allows researchers to plug-in their planners. Because EUROPA was developed by the NASA to support specific missions, the planner is intended to be domain-dependent. The EUROPA utilises a New Domain Definition Language (NDDL) (Barreiro et al., 2012), an implementation of Constraint-based Attribute and Interval Planning (CAIP) (Frank & Jónsson, 2003) paradigm. Another of NASA's framework is the ASPEN (Automated Scheduling and Planning ENvironment) (Chien et al., 2000). Importantly, the ASPEN performs *iterative search and repair* in contrast to depth-first search spread in the other planners.

The European Space Agency's response to the timelines planning is an Advance Planning and Scheduling Initiative (APSI) (Simone Fratini & Cesta, 2012) software platform which implements Timeline-based Representation Framework (TRF) (Cesta & Simone Fratini, 2008)). The TRF is an equivalent to the EUROPA. Hence, it is not a planner, but it provides a unified tool in how to manage a search space for a planner on top of the TRF's hierarchy. The TRF can be exploited by any planner. For example, MrSPOCK (Cesta, Cortellessa, Fratini & Oddi, 2009) (Mars Express Science Plan Opportunities Coordination Kit) was used by the ESA to produce long-term – about 4 months – skeleton plans for a Mars spacecraft that were then adjusted by human operators when needed.

9.1.2.3. Planners with a Single Constraint Network

The OMPS (Open Multi-component Planner and Scheduler) (S. Fratini et al., 2008) obeys domain synchronisation rules by making a decision on a component. A component is a generalisation of a state variable in order to handle different constraints in the same way. The lessons learned from the OMPS have been used in designing the Meta-CSP framework (Rocco, Pecora & Saffiotti, 2013b). This work is motivated by the following observation. Ordering decisions in a constraint network can result in multiple choices. For example, a token can be ordered before or after another one to resolve their resource conflict. The Meta-CSP framework does not choose between decisions. Instead, it builds a CSP network for *decision variables*, on top of this standard network. For each decision variable, its finite domain consists of alternative bottom constraint networks that accommodate one of the possible decisions. Hence, all the choices are preserved. This planner is further extended into the planner CHIMP (Conflict-driven Hierarchical Meta-CSP Planner) (Stock et al., 2015) in order to utilise hierarchical planning (HTN) as a tool to limit the expansive search space.

9.1.2.4. Summary

Older planners such as IxTeT, the EUROPA, the ASPEN and the MrSPOCK utilise multiple separate networks. In contrast, most of the recent planners, such as the OMPS, the Meta-CSP framework and the CHIMP exploit a heterogeneous approach where different constraints are modelled as one type. The clear benefit of this approach is that it can be easily extended to accommodate for other types of constraints. This is especially beneficial in robot domains where such a timeline planner can reason at the same time about tasks, the robot's path, a trajectory of its manipulators, a spatial distribution of objects, etc., as demonstrated by Mansouri and Pecora (2014a).

9.1.3 Summary

State-based and time-based planners strongly differ:

- in the underlying representation. On the one hand, time-based representation via timelines is well suited to model complex real-world scenarios, as illustrated by the NASA and ESA applications. On the other hand, state-based representation supporting durative actions has evolved from classical planning. Therefore, it preserves the notion of static states, which are transitioned by effects of actions relating to time instances.

- in their search space. State-based planners represent a search state as a conjunction of literals describing a state of the modelled world. In contrast, timelines planners search in the space of constraint networks, i.e., partial plans.

- in a found solution. State-based planners optimise their solution, i.e., they provide a single best plan, whereas timelines planners provide a set of plans without considering their quality. It is an open question how an optimisation criterion in timeline planning should be formalised. The main difficulty is how to measure a quality of a set of plans rather than a single plan.

We can summarise this comparison by stating that while timeline representation captures different aspects of real world scenarios well, solving of such models is challenging from a computational point of view (Benedictis & Cesta, 2015).

Timeline representation seems to be more relevant to our domain due to the NASA and ESA applications. However, as we intend to integrate an existing planner into our algorithm, the performance of the chosen planner is critical to us. The planner should not only be reasonably fast but also provide a good quality of solutions. The quality of plans is needed as we assume that a mobile service robot can be oversubscribed by tasks. Therefore, if the robot can optimise its plan, it can achieve as many tasks as possible. As a result, we are interested in comparing performances of state-based and time-based planners. To our best knowledge, there is no survey providing such a comparison. We report at least on a few findings from different research papers.

Bit-Monnot (2016) has compared the FAPE with three state-based planners: POPF (A. J. Coles et al., 2010), OPTIC (Benton et al., 2012) and TFD (Eyerich et al., 2009) on several domains from the International Planning Competition (IPC). Unfortunately, it has compared only their coverage on these domains. The FAPE has comparable coverage with POPF and OPTIC, but TFD has significantly better coverage (solving 183 problems, whereas the FAPE solved only 114 across 12 domains).

Benedictis and Cesta (2015) compare runtime of their proposed timeline planner iLOC to runtimes of the VHPOP (Simmons & Younes, 2011), OPTIC and COLIN (Amanda Jane Coles, Coles, Fox & Long, 2012). They compare runtime of the planners as they focus on developing such a heuristic that would speed up a search of timeline planners. The evaluation has been made using a number of domains from the IPC. They have demonstrated that their proposed heuristic can speed up the search of their timeline planner but the tested state-based planners are still significantly faster on most of the domains.

To conclude, based on this comparison, state-based planning is more promising for our application because of its speed, coverage and the notion of plan quality. Additionally, limitations of PDDL do not restrict modelling of our domain. However, before we make a final decision on the representation to use, we review plan merging techniques for both representations in the following chapter.

9.2 Plan Merging

Given a set of n input plans, where each plan achieves a single goal G_i, plan merging composes a single plan that achieves all goals $G_1, \dots G_n$ by reusing parts of the input plans. Plan merging is often questioned because of the analysis done by Nebel and Koehler (1995). The authors analytically proved that *plan reuse* cannot lead to provable efficiency gain, compared to planning from scratch. However, *plan reuse* and *plan merging* are very different problems. Plan reuse composes a plan to achieve a new goal G by *reusing* plans from a database. Importantly, these plans achieve different goals than G. Hence, reasoning about related actions from the database plans has, in theory, the same complexity as the plan generation. However, this argument cannot be reapplied to the problem of plan merging because plan reuse is a very different problem. Although this claim does not prove that plan merging has a smaller complexity than plan generation. The complexity analysis for a merging problem is not presented in the literature, to our best knowledge, and it is outside of the scope of this thesis.

9.2.1 Action Merging

The term "plan merging" first occurred in the system NOAH (Sacerdoti, 1975), followed by NONLIN (Tate, 1976) and SIPE (Wilkins, 1988). However, in these systems, "plan merging" referred only to the changes done to a single plan in order to optimise it. Foulser, Li and Yang (1992) define "plan merging" as *"an important type of helpful goal interaction occurs when certain operators in a plan can be grouped, or merged, together in such a way as to make the resulting plan more efficient to execute."* Such techniques should be denoted as *action merging*. Therefore, we strictly differentiate between plan merging and action merging in this literature review.

9.2.2 Plan Merging as Refinement planning

In contrast to forward-chaining planners, *refinement planning* is initialised with a set of actions with no constraints between them (Kambhampati, Knoblock & Yang, 1995). A solution plan is reached by adding refinements based on interactions between the actions. These refinements limit the possible action orderings. Krogt (2005) extended refinement planning in order to consider also *unrefinements*, i.e., that not all actions need to be used. This is particularly useful for action merging, where an action can be removed from a plan when its effects are satisfied by another action.

Plan merging via refinements can be modelled either as a satisfiability problem (Mali, 1999) or as a Constraint Satisfaction Problem (CSP) (Yang, 1997). To our best knowledge, (Yang, 1997) is the first work to introduce plan merging. Therefore, we provide more detail about this work. Yang's algorithm first creates a piecewise union of the input plans and then finds all *conflicts* between them. A conflict is when an effect of an action may change a precondition required by another action, i.e., when an action from one plan violates causality of another plan. Therefore, a refinement needs to be added to order these actions appropriately. The action with the conflicting effect can be ordered:

- after the action requiring the precondition (*promotion*);
- before the action requiring the precondition (*demotion*);

Conflicts are resolved by a Constraint Satisfaction Problem (CSP). Hence, all conflicts are found first in order to build a CSP. Then, a conflict represents a node and the domain of each node is its possible resolution: promotion or demotion. This CSP can be solved by any standard algorithm.

Tsamardinos, Pollack and Horty (2000) extends Yang's work in order to merge temporal plans. Because it has been introduced before PDDL 2.1 was proposed, the authors introduce their own representation of time. Important to us is that their representation follows the *temporally-lifted approach*. Hence, an action is represented as two instances – the start and end– which are constrained by the action duration. Because actions are durative, they may be ordered in parallel. Therefore, another resolution of a conflict is possible:

- the action with the conflicting effect can be ordered not in parallel to the action requiring the precondition, hence both orderings (before or after) are possible (*separation*).

In order to model *separation* ordering, the authors use a Conditional Simple Temporal Network (CSTN), which extends Simple Temporal Networks (STNs) (Dechter et al., 1991) by supporting disjunction between two possible orderings. A CSTN is used to detect conflicts and to check the temporal consistency of a solution proposed by an underlying CSP (as in Yang's work).

Formulating plan merging as planning with refinements has two main disadvantages:

- the input plans must be compatible, i.e, there exists such an order on actions that guarantees a conflict free plan.

- the final plan is not optimised. The CSPs provide a valid solution, but they do not evaluate how good the solution is.

The first drawback is a significant bottleneck of this approach. There can be many situations where two plans cannot be merged due to their conflicting structure.

9.2.3 Plan Merging as Plan Repair

The requirement of refinement planning that plans need to be compatible is overcome by the usage of plan repair techniques. Originally, plan repair techniques were proposed in order to save a planning effort when execution of a plan fails. Instead of discarding the failed plan and planning from scratch, plan repair techniques *repair* the plan, i.e., the part of the plan that has caused the failure is replaced by another plan (Bidot, Schattenberg & Biundo, 2008).

Harris and Dearden (2012) address plan merging via plan repair in a domain of an oversubscribed underwater robot. In their work, a robot executes a plan that is guaranteed not to violate the limit of a robot's battery. During plan execution, the battery level is monitored. When there is enough energy left, a robot can opportunistically merge a plan for another goal into the existing plan. This means that the original plan needs to be *repaired* in order to accommodate the new plan.

9.2.4 Plan Merging as Plan Coordination

Plan merging can also be seen as a problem of *plan coordination* between multiple robots (Ephrati & Rosenschein, 1993), (de Weerdt, 2003). Plan coordination combines planning with refinements and plan repair. Plan coordination is important in domains where multiple robots with different capabilities need to cooperate in order to achieve given tasks (Tonino, Bos, de Weerdt & Witteveen, 2002). In a plan coordination problem, each robot has its separate plan, but the plans need to be synchronised by imposed orderings and to be *repaired* in order to allow coordination between the robots. Finally, coordination of temporal plans is addressed by Hashmi and El Fallah Seghrouchni (2010) and Allouche and Boukhtouta (2010).

9.2.5 Plan Merging with Timelines

Merging of timeline plans can be viewed as plan coordination as well. Because a timeline plan is represented by a constraint network, plan merging narrows down to finding conflicts in a bigger network created by conjoining the input networks (i.e., plans). A conflict can be resolved by either:

- imposing ordering constraints between conflicting tokens (similarly as in plan merging via refinements); or
- repairing the network by adding new tokens.

Recall that these two techniques are the same as in the case of normal timeline planning, as discussed in Sec. 9.1.2.1.. Therefore, plan merging is technically not different to planning in timeline representation. However, Stock et al. (2015) demonstrate that their proposed CHIMP planner can still benefit from merging of separately generated plans, compared to conjoined planning for a single large problem.

9.3 Chosen Representation for Plan Merging

Finally, we can decide what representation of temporal planning is going to be used for the rest of this thesis. Recall that we have argued that state-based planning seems more promising to us due to its speed, coverage and quality of plans. In addition to this, we have discussed that algorithms for plan merging in timeline planning are not any different from algorithms for planning. Hence, plan merging problem is very uninteresting in timeline planning. In contrast, merging in temporal state-based planning raises a number of interesting issues:

- how a *separation* resolution of conflicts can be represented and utilised;
- how to preserve the temporal relation between the start and end action instances;
- how to preserved required concurrency in the input plans.

Moreover, our assumptions **A5**-**A7** add more aspects which must be considered while merging in state-based planning. Based on all these arguments, we have decided to focus on state-based planning. To conclude, when we refer to *temporal planning* from now on, we mean strictly state-based temporal planning assuming limitations imposed by PDDL 2.1 level 3. Our framework currently supports neither ADL nor numeric variables.

CHAPTER 10

TERMINOLOGY

This chapter formally defines the nomenclature used throughout this thesis.

10.1 State

A state is represented by a conjunction of *literals* L_s where a literal l is either an atomic formula (atom) f or its negation $\neg f$; f and $\neg f$ cannot be presented at the same state as they are exclusive to each other. Atomic formulas F describe a certain property of the modelled world. They are expressed as a *predicate* followed by *terms*. For example, a formula (at r1 loc1) means that the robot r1 is at location loc1. A collection of all atoms describing the world properties is called a domain $D = \{f_1, .., f_n\}$, hence all possible literals to describe a state are $L_D = \{f_1, \neg f_1, \ldots, f_n, \neg f_n\}$ and it holds that $L_s \subset L_D$. Additionally, we define satisfied and unsatisfied literals with regard to a state s.

Definition 10.1.1. ***Satisfied literal:*** *Literal l is satisfied in the state literals L_s if $l \in L_s$.*

Definition 10.1.2. ***Unsatisfied literal:*** *Literal l is unsatisfied in the state literals L_s if $l \notin L_s$.*

10.2 Action

10.2.1 Instantaneous Action

The modelled world transitions from one state to another by an instantaneous *action* a from a set of all possible actions A. Action $a = \langle L^{pre}, L^{eff} \rangle$ is described by its precondition literals L^{pre} and effect literals L^{eff} . In instantaneous planning, preconditions $L^{pre} \subset L_D$ are a set of literals that must be *satisfied* in a state s in order to transit it by the action a to another one. Conversely, effects $L^{eff} \subset L_D$ are a set of literals that are applied by the action after a transition is done. Two mapping functions, which return the set of preconditions or effects, are defined as follows.

Definition 10.2.1. ***Precondition function:*** *A function $pre : A \rightarrow L_D$ returns for an action $a \in A$ its preconditions $L^{pre} \subset L_D$.*

Definition 10.2.2. ***Effect function:*** *A function $eff : A \rightarrow L_D$ returns for an action $a \in A$ its effects $L^{eff} \subset L_D$.*

Finally, equality of two actions is defined as follows.

Definition 10.2.3. ***Equal actions:*** *Action a_1 is equal to action a_2 if and only if $pre(a_1) = pre(a_2)$ and $eff(a_1) = eff(a_2)$.*

10.2.2 Durative Action

The above definition of an action is extended by processing time (duration) $p \in \mathbb{R}$, which represents how long the action lasts in a continuous time space. Therefore, action a is represented by a tuple $a = \langle L^{pre}, L^{eff}, p \rangle$. Additionally, the definition of equality (Def. 10.2.3), is extended by adding the requirement that durations of the actions a_1 and a_2 must be equal too, i.e., $p_1 = p_2$. Moreover, temporal state-based planning splits the action into two instances: the start and end point of the action. Therefore, the preconditions and effects can hold *at the start*, referred to as $L_{\vdash}^{pre}$ and $L_{\vdash}^{eff}$, respectively; and *at the end* of the action, denoted as $L_{\dashv}^{pre}$ and $L_{\dashv}^{eff}$; Additionally preconditions can be required to hold *over all* duration of the action, $L_{\leftrightarrow}^{pre}$. These preconditions are referred to as *invariants*. Effects holding throughout the action are the same as effects at the start, hence they are omitted. Therefore, the preconditions and effects can be split into groups containing only the specific literals, such that $a = \langle \{L_{\vdash}^{pre}, L_{\leftrightarrow}^{pre}, L_{\dashv}^{pre}\}, \{L_{\vdash}^{eff}, L_{\dashv}^{eff}\}, p\rangle$.

Finally, given action $a = \langle \{L_{\vdash}^{pre}, L_{\leftrightarrow}^{pre}, L_{\dashv}^{pre}\}, \{L_{\vdash}^{eff}, L_{\dashv}^{eff}\}, p\rangle$, the action start and end points can be defined, as an instantaneous action and a time stamp.

Definition 10.2.4. ***An action start point*** *$a_{\vdash} \in A \times \mathbb{R}$ is a pair $(\langle \{L_{\vdash}^{pre}, L_{\leftrightarrow}^{pre}\}, L_{\vdash}^{eff}, 0\rangle, t_{\vdash})$ where $t_{\vdash}$ is a time stamp representing when action a starts to alter the current world state. Accordingly, we define a set of actions' start points as $A_{\vdash}$.*

Definition 10.2.5. ***An action end point*** *$a_{\dashv} \in A \times \mathbb{R}$ is a pair $(\langle \{L_{\leftrightarrow}^{pre}, L_{\dashv}^{pre}, \}, L_{\dashv}^{eff}, 0\rangle, t_{\dashv})$ where $t_{\dashv} = t_{\vdash} + p$ is a time stamp representing when action a ends. Similarly, we define a set of actions' end points as $A_{\dashv}$.*

Definition 10.2.6. ***Preconditions of action point*** *are denoted as $pre(a_{\vdash}) = pre(a)_{\vdash} \cup \{pre(a)_{\leftrightarrow} \setminus eff(a)_{\vdash}\}$; and likewise for the effects of the action point.*

10.2.3 Instantaneous Action as an Action Point

The definition of an action point can be reused in re-defining an instantaneous action as an action point. We propose such a definition because we are going to introduce terms that hold not only for instantaneous actions but also for action points. Therefore, in order to avoid giving two very similar definitions for each term, we unify an instantaneous action with definition of an action point.

Definition 10.2.7. ***Instantaneous action as an action point:*** *An action point $a_{\perp} \in A \times \mathbb{R}$, which represents an instantaneous action a, is a pair $(\langle L^{pre}, L^{eff}, 0\rangle, \emptyset)$. We denote the set of instantaneous actions as $A_{\perp}$.*

Definition 10.2.8. ***Arbitrary action point*** *is denoted by the symbol a_{H}. Importantly, in an instantaneous domain, a_{H} stands for an instantaneous action point. However, in a temporal domain, it stands either for an action start or an end point.*

Definition 10.2.9. ***Action points:*** *In an instantaneous domain, all instantaneous action points correspond to the set of all actions A. But in temporal planning, all possible actions' points A_{H} consists of start and end points, that is $A_{\mathsf{H}} = A_{\vdash} \cup A_{\dashv}$.*

10.3 Distinctive Actions

Some actions have unique roles when perceived in the broader content of a state transition. In order for atom f to hold in state s, it must be *achieved* by an action. Such an action is referred to as the *achiever* of atom f. Vice versa, for atom f to not hold in state s, it is *deleted* by an action that is called the *deleter* of atom f. Instead of following the standard approach of defining achievers and deleters as actions, we define them as action points. This extension is made because atoms are changed due to applying the effect belonging to a specific action point.

Achievers and deleters utilise the following notion of positive and negative preconditions and effects.

Definition 10.3.1. ***Positive and negative preconditions:*** *The positive precondition literals L^{pre^+} of action a are returned by a function $pre^+ : a \rightarrow F$. In contrast, the negative precondition literals L^{pre^-} of action a are returned by a function $pre^- : a \rightarrow \neg F$.*

Using this definition, the achievers and deleters are defined as follows.

Definition 10.3.2. ***Achiever of an atom:*** *When a plan is performed, effects of action points from the plan affect the current state L_s. For a positive literal $f \in L_s$, the last performed action point having f as its effect is called an* achiever *of atom f, denoted as a_f^+. Conversely, for a negative literal $\neg f \in L_s$, the last performed action point having $\neg f$ as its effect is called a* deleter *of atom f, denoted as a_f^-.*

A state transition is obtained by applying the effects of an *applicable* action point.

Definition 10.3.3. ***Applicable and inapplicable action point in a state:*** *Action point a_{H} is applicable in state s if and only if its preconditions hold in the state, hence $pre(a_{\mathsf{H}}) \in L_s$. We denote the set of all applicable action points in state s as $\mathcal{A}^{app}$. The set of all inapplicable action points in state s is $\mathcal{A}^{inapp} = \mathcal{A}_{\mathsf{H}} \setminus \mathcal{A}^{app}_{\mathsf{H}}$.*

In the case of temporal planning, an end action point $a_{\dashv}$ is dependent on the start action point $a_{\vdash}$ of the same action a. Hence, its applicability needs to be extended in the following way. Action a transits state s_1 to state s_2 if and only if $a_{\vdash}$ is applicable in state s_1 at time $\tau_{\vdash}$ and if there is a guarantee that $a_{\dashv}$ will be applicable at time $\tau_{\dashv} = \tau_{\vdash} + p$, where p is duration of action a. This means that either:

(S1) $pre(a)_{\dashv}$ must be satisfied in state s_1 and must keep holding until time $\tau_{\dashv}$; or

(S2) $pre(a)_{\dashv}$ are not satisfied in s_1 and for each literal $l \in pre(a)_{\dashv}$, there is an achiever a_l^+ which has literal l as its effect. As a result, the literal l holds from time τ^+ (when the achiever effect was applied) until the time $\tau_{\dashv}$ when it is needed.

These two situations are described in the example below.

Additionally to these conditions, a durative action a also requires that its invariant preconditions hold throughout its duration. We refer to such literals as *protected.*

Definition 10.3.4. ***A protected literal*** *l occurs if and only if: (i) literal l is in the invariant preconditions of a, i.e., $l \in pre(a)_{\leftrightarrow}$; (ii) the action's start point $a_{\vdash}$ has already been applied; (iii) the action end point has not been applied, i.e., action a is not yet finished. We refer to the pair of the protected literal and the related action point as $l^{prot} = (l, a_{\vdash})$.*

Note that this may remind *threatened* literals in causal links. A threatened literal is in a causal link between two actions' points from two *different* actions. In contrast, a *protected* literal is a link between a start point and an end point of the same action. Furthermore, only invariant timed literals ("over all") can be protected but any timed literal can be threatened. A protected literal can be threatened similarly as a threatened literal by another action point a'_{H} that has $\neg l$ as its effect. However, the solution is possible only by *promotion* due to the fact that the action, causing the protected literal, has already started. Hence a'_{H} cannot be ordered before it, i.e. demotion cannot be applied.

10.3.1 Example

The introduced nomenclature can be illustrated by the following example. A temporal action a with duration p_a has three preconditions, where the first needs to hold at the *start* of the action, the second needs to hold *over all* duration and the third in the *end*. Additionally, it has one effect at the end. Fig. 10.1a illustrates the situation S1 from the above list. Because both $pre(a)_{\vdash}$ and $pre(a)_{\leftrightarrow}$ hold in the current state s_1, the start action point $a_{\vdash}$ is *applicable* in state s_1. Therefore, the action point is applied to the state at time $\tau_{\vdash}$. As a result, the invariant literal $l \in pre(a)_{\leftrightarrow}$ becomes *protected* until the end action point is applied. When time $\tau_{\dashv} = \tau_{\vdash} + p_a$ is reached, the end action point needs to be applied. Because the precondition $pre(a)_{\dashv}$ holds, this action point can be applied, releasing the protected literal. Furthermore, this end action point is also an *achiever* to the literal $l \in \mathit{eff}(a)_{\dashv}$. As this literal has become true at the end of action a, the state has changed into s_2. Additionally, any *deleter* of $pre(a)_{\leftrightarrow}$ has been *promoted* to occur *after* the end of the action a. In the figure, the at end effect of the deleter a^{-} has changed the literal in $pre(a_{\leftrightarrow})$ to false (F). It is important that such a change does not occur not in the interval $\vartheta_1 = (\tau_{\vdash}, \tau_{\dashv})$, where the literal was *protected* to be true by the action a.

Situation S2 from the above list is illustrated in Fig. 10.1b. The start action point $a_{\vdash}$ is *applicable* in state s_1. However, the end action point $a_{\dashv}$ is not, because

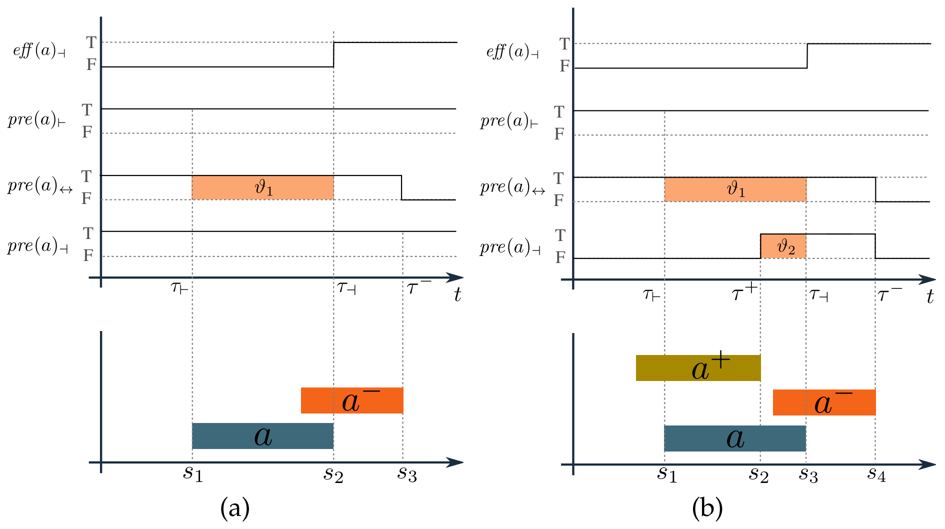


Figure 10.1: The action a has three literals as the preconditions and one effect. In (a), all the preconditions are true (T) at the state s_1 and any deleter of the literals, making them false (F), is ordered after they are needed, such as deleter a^- of $pre(a)_{\leftrightarrow}$ appearing at time $\tau^- > \tau_{\dashv}$. The interval ϑ_1 highlights where the invariant precondition is protected. In (b), the precondition $pre(a_{\dashv})$ is false in s_1 but the achiever a^+ makes this literal true at the time $\tau^+ < \tau_{\dashv}$. Moreover, this literal may be threatened within the interval ϑ_2, hence any deleter must occur outside of this interval.

the precondition is false at the state s_1. Hence, a planner must guarantee that there is an *achiever* that makes the literal true at time τ^+ that is before $\tau_{\dashv}$, when the *at end* literal is needed. In our case, the end point of action a^+ achieves the literal. Moreover, there cannot be an effect deleting this literal in the interval $\vartheta_2 = (\tau^+, \tau_{\dashv})$. The literal is *threatened* on this interval, hence all deleters must be either *demoted* to be before τ_+ or promoted after $\tau_{\dashv}$.

10.4 Plan

An ordered set of action points, which transits the world from an *initial* state $I \subset L_D$ to a required *goal* state $G \subset L_D$, is called a *plan*, denoted as π. State s with a set of literals L_s *achieves* the goal if $G \subseteq L_s$. The pair $P = (I, G)$ is referred as a *planning problem*, and a plan π is its solution. In order to measure the *quality* of a solution, the notion of *makespan* of the plan is used. Whilst in instantaneous planning makespan is expressed as the number of actions in the plan, in temporal planning, makespan is expressed as the duration of the plan. As actions can occur concurrently in the plan, plan duration cannot be defined by simple summation of their duration.

Definition 10.4.1. ***Plan quality*** *is* $Q = \frac{1}{|\pi|}$ *where* $|\pi|$ *is makespan of plan* π*, which is measured as* $|\pi| = t_{\dashv}^{END} - t_{\vdash}^{START}$*. The symbol* $t_{\dashv}^{END}$ *refers to the time when the last action* ***END*** *ends and* $t_{\vdash}^{START}$ *is the time when the first action* ***START*** *starts.*

The actions START and END are special actions that are utilised in the representation of a plan.

Definition 10.4.2. ***The START action*** *is the first action in the plan such that* $\langle \emptyset, I, 0 \rangle$*. This means that it is an instantaneous action point, whose preconditions are empty and effects are set to the initial state* I*.*

Definition 10.4.3. ***The END action*** *is the last action in the plan such that* $\langle G, \emptyset, 0 \rangle$*. This means that it is an instantaneous action point, whose preconditions are set to be the goals literals* G *and effects are set to be empty.*

10.4.1 Partial Order Plan

In this thesis, we exploit *partially ordered plans* (POP) that are already *grounded*[1]. For a thorough overview of POP see (Weld, 1994).

[1]Ground action means that specific values are assigned to all terms in the literals, hence there are no variables left. Therefore, in this definition, we omit details about lifted actions and bindings when describing the plan representation.

Definition 10.4.4. ***A partially ordered plan (POP)*** *is a tuple $\pi = \langle \mathcal{A}_{\mathsf{H}}, L, O \rangle$, where:*

- *$\mathcal{A}_{\mathsf{H}}$ is a set of action points such that their actions are a subset of all possible actions, $\mathcal{A} \subseteq A$;*
- *$\mathcal{L}$ is a set of causal links; a causal link links two action points from $\mathcal{A}_{\mathsf{H}}$ together in a sense of producing literal l and consuming it, see exact definition below.*
- *$\mathcal{O}$ is a set of ordering constraints between two action points from $\mathcal{A}_{\mathsf{H}}$ defining a partial order on the set $\mathcal{A}$.*

Definition 10.4.5. ***A causal link*** *$a_{\mathsf{H}} \xrightarrow{l} a'_{\mathsf{H}}$ represents that literal $l \in pre(a'_{\mathsf{H}})$ is produced by an effect of action point a_{H}, i.e. $l \in eff(a_{\mathsf{H}})$. We refer to a_{H} as a* producer *of literal l in order to highlight that the literal is produced for some other action point. Additionally, we refer to a'_{H} as a* consumer *of literal l in order to highlight that this action point consumes the literal.*

Definition 10.4.6. ***Ordering:*** *The relationship between two action points $a_{\mathsf{H}}\, o\, a'_{\mathsf{H}}$ from different actions is specified by an ordering $o \in O$ where $O = \{<, =, >, \emptyset, \leq, \geq\}$. The ordering $\emptyset$ means that no ordering is defined, hence the two action points can appear in any order. The rest of the symbol has the standard meaning. Noticeably, the ordering $\neq$ is not supported as it would lead to a disjunction of two possible orderings. An extension, such as this, would lead to contingent planning, which is outside the scope of this thesis. Additionally, there is always ordering $a_{\vdash} < a_{\dashv}$ for a durative action a.*

10.4.2 Backward-Chaining Plan-Space Planning

Historically, POP algorithms were designed as a backward search through the space of partial plans. An initial plan in such a case contains only START and END actions and this plan is expanded by adding actions to resolve unsatisfied preconditions of actions already in the plan - this is known as *an open condition* flaw.

Moreover, these actions need to be ordered to avoid another flaw – *a threatened causal link.*

Definition 10.4.7. ***An open condition*** $\xrightarrow{l} a'_{\mathsf{H}}$ *means that literal* $l \in pre(a'_{\mathsf{H}})$ *has not yet been linked to an effect of any producing action point.*

Definition 10.4.8. ***A threatened link*** *is a causal link* $a_{\mathsf{H}} \xrightarrow{l} a'_{\mathsf{H}}$ *such that there is an action point* $a''_{\mathsf{H}} \in \mathcal{A}_{\mathsf{H}}$ *that could possibly be ordered between* a_{H} *and* a'_{H} *and threatens the causal link by having* $\neg l$ *as its effect, i.e.,* $\neg l \in \mathit{eff}(a''_{\mathsf{H}})$.

Open conditions are solved by adding new action points to $\mathcal{A}_{\mathsf{H}}$ and by creating new causal links saved in $\mathcal{L}$. Threatened links are addressed by adding new orderings to $\mathcal{O}$ to make sure that the threatening action point a''_{H} does not occur between the action point linked in the threatened link $a_{\mathsf{H}} \xrightarrow{l} a'_{\mathsf{H}}$.

Definition 10.4.9. ***Promotion and demotion*** *are techniques which solve a threat to a causal link* $a_{\mathsf{H}} \xrightarrow{l} a'_{\mathsf{H}}$ *from* a''_{H}. *Only one technique is applied to solve the threat.* Promotion *adds the constraint* $a'_{\mathsf{H}} \prec a''_{\mathsf{H}}$ *to* $\mathcal{O}$. *Conversely,* demotion *adds* $a''_{\mathsf{H}} \prec a_{\mathsf{H}}$ *to* $\mathcal{O}$.

10.4.3 Required Concurrency

Required concurrency is an important aspect in temporal planning that describes that some actions are required to be ordered in a certain relation. It can be illustrated on the example from the *turn and open* domain, where a robot with a gripper is required to open a door. In order to do so, it needs to turn the door knob and hold it turned whilst it pushes the door. Hence, the action push is required to be ordered *during* action turn. The synchronisation of actions is critical for the success.

This property is defined by the following definition, which is slightly rephrased from the one given by W. Cushing et al. (2007).

Definition 10.4.10. ***Required concurrency:*** *Let a and a' be durative actions from a plan. The actions are concurrent if either $a_{\vdash} \leq a'_{\vdash} \leq a_{\dashv}$ or $a'_{\vdash} \leq a_{\vdash} \leq a'_{\dashv}$. A plan is concurrent when any action is concurrent, otherwise the plan is sequential. A solvable planning problem has* required concurrency *when all solutions are concurrent.*

10.5 Task

Throughout this thesis, we have used the examples of tasks given to the robot, such as:

- ω_1: "notify people in a common eating area to leave between 20:00 - 20:15"

This informal description maps into a formal one in the following way. The command "notify people in a common eating area to leave" is translated into a goal G. Additionally, the robot is aware of the current state of its environment. This knowledge is expressed as an initial state I. The goal and the initial state are specified by a conjunction of literals from the domain D. Furthermore, the initial state and goal state represent a planning problem $P = (I, G)$. A solution to this problem is a plan π that is a set of ordered actions from all robot's capabilities A, such as navigating to the eating area and interacting with people.

The initial state I is presented because planning is traditionally formalised as searching for a sequence of actions transitioning the world from the initial state to the goal state.

10.5.1 Task Instances

The task described above is defined as follows.

Definition 10.5.1. ***An instantaneous task*** *is defined as a tuple $\omega_i = \langle D_i, A_i, (I_i, G_i), \pi_i\rangle$ containing the task domain D_i, the set of instantaneous actions A_i, the planning prob-*

lem $P_i = (I_i, G_i)$ and the partially ordered plan π_i, which solves the planning problem using the given domain and actions.

However, such a task is unrealistic in the real world. Therefore, we introduce a durative task, which operates with durative actions. As a result, the plan lasts for a certain amount of time. Its duration corresponds to the duration of the whole task. In the previous part of the thesis, we denoted the task duration as the *processing time* p. Thus, we will keep this notion here as well.

Definition 10.5.2. *A* ***durative task*** *is defined as a tuple $\omega_i = \langle D_i, A_i, (I_i, G_i), \pi_i, p_i \rangle$ containing the task domain D_i, the set of durative actions A_i, the planning problem $P_i = (I_i, G_i)$, the partially ordered plan π_i and the duration of the plan p_i.*

Finally, we can re-introduce the *release date* r and the *deadline* d from the previous part. As before, we argue that *all tasks* must be within the time window limited by the release date and deadline.

Definition 10.5.3. ***A durative task with a time window*** *is defined as a tuple $\omega_i = \langle D_i, A_i, (I_i, G_i), \pi_i, p_i, r_i, d_i \rangle$ containing the task domain D_i, the set of actions A_i, the planning problem $P_i = (I_i, G_i)$, the partially ordered plan π_i, the duration of the plan p_i, the release date r_i and the deadline d_i. Additionally, the plan must fit the given time window limited by the release date and deadline.*

10.5.2 Task Dependency

One of the introduced requirements in this thesis was that the given tasks must be separable. A task is separable if there exists no other task that must precede it or follow it. However, we argued that spatial and temporal constraints cause dependency between their sub-parts. We elaborate on this phenomenon here in detail, where we express task dependency only according to causal constraints between them. We omit temporal constraints as the causality is a stronger requirement when compared to temporal constraints. For example, assume two tasks, whose time windows are overlapping. They are interacting in time but they may be no causal constraints between them, i.e., they can be ordered in parallel.

Definition 10.5.4. ***Independent tasks:*** *Two tasks ω_i, ω_j are independent of each other if actions from their plans π_i and π_j can be performed in any order.*

Definition 10.5.5. ***Semi-dependent tasks:*** *Two tasks ω_i, ω_j are semi-dependent on each other if* some *actions in their plans π_i, π_j must be performed in a certain order.*

Definition 10.5.6. ***Dependent tasks:*** *Two tasks ω_i, ω_j are dependent on each other if* all *actions in their plans π_i, π_j must be performed in a certain order.*

CHAPTER 11

PROBLEM DEFINITION AND SOLUTION APPROACHES

In this part of the thesis, we are interested in finding a single plan that satisfies all given tasks whilst meeting their spatial and temporal constraints. In order to be able to compare our proposed solution with the existing planners, we introduce two simplified problems. The first simplified problem assumes that given tasks are *instantaneous* and there are no temporal constraints. The second simplified problem operates with *durative* tasks but still without temporal constraints.

11.1 Problem Statement

The problem we are interested in is a composition of a single, partially ordered plan (POP) from a given set of input tasks that can be dependent according to Def. 10.5.6. This means we want to find a plan π that achieves goals $G_1, ..., G_n$ of the given set of tasks $\Omega = \{\omega_1, \dots, \omega_n\}$. We investigate three task types, in increasing level of complexity:

1. instantaneous tasks, i.e., $\omega_i = \langle D_i, A_i, (I, G_i), \pi_i \rangle$,

2. durative tasks, i.e., $\omega_i = \langle D_i, A_i, (I, G_i), \pi_i, p_i \rangle$, and

3. durative tasks with time windows, i.e., $\omega_i = \langle D_i, A_i, (I, G_i), \pi_i, p_i, r_i, d_i \rangle$.

The initial states of each task are mutually consistent. This means that I_j is consistent with I_k if for all $l \in L_{D_j} \cap L_{D_k}$, $l \in I_j$ if and only if $l \in I_k$.

11.2 Solution Approaches

In Sec. 8, we have illustrated three different approaches addressing this problem: *sequencing*, *conjoining* and *merging*. In this section, we formally define them.

11.2.1 Sequencing planning algorithm

This approach generates an independent plan for each task ω_i. Hence, it finds a set of plans $\Pi_\omega = \{\pi_1, \ldots, \pi_n\}$, and then *sequences* them to create a single final plan π_f. For the resulting plan to be valid, one needs to first decide on an ordering of the tasks and then modify the initial states of each task to the final state of the plan for the preceding task. This approach is generally very fast, but it comes at the price of plan quality because this approach does not allow for the interleaving of actions from different plans. This means it does not take advantage of possible synergies between tasks.

11.2.2 Conjoining planning algorithm

This approach relies on *conjoining* tasks from the input set into a single task (i.e., a single planning problem). This is possible for instantaneous tasks and durative tasks with no constraints. The conjoined instantaneous task is defined as:

$$\omega = \left\langle \bigcup_{i \in \{1 \ldots n\}} D_i, \bigcup_{i \in \{1 \ldots n\}} A_i, \left(\bigcup_{i \in \{1 \ldots n\}} I_i, \bigcup_{i \in \{1 \ldots n\}} G_i \right), \pi_f \right\rangle \tag{11.1}$$

and similarly for the temporal tasks. After a conjoined task is obtained, one can use an appropriate planning algorithm to find its solution plan π_f. While this approach can more easily take advantage of the relationships between goals in different tasks in comparison to sequencing (e.g., two tasks that should be executed in the same location), it can suffer from scalability issues as finding a plan for the conjoined task can be much harder than finding plans for each individual task by itself. In this work, we assume that the conjoined tasks are modelled a priori and passed directly as an input to such systems. However, the release dates and deadlines cannot be conjoined in this way. We elaborate on how they are conjoined in Chapter 13.

11.2.3 Merging planning algorithm

This approach combines both of the aforementioned methods. It also plans for tasks separately, obtaining $\Pi_\omega = \{\pi_1, \dots, \pi_n\}$, but then it reasons over each plan and *merges* them together into a single plan with the same or better quality, compared to the plan obtained by simple sequencing. The final plan π_f consists of parts of the task plans in Π_ω and newly created plans Π_{join} by *plan repair* techniques. These plans are used in order to connect the parts of Π_ω such that the final plan is free of flaws.

While the merging procedure adds an overhead at plan generation time, when compared to the sequencing approach, it allows us to find synergies and demands between plans for different tasks, which results in interleaving plans for different goals. A typical benefit of this approach in the mobile robot domain is the possibility to perform actions from different tasks when these actions share a common location.

CHAPTER 12

SOLUTION

We propose a novel algorithm, POPMER$_X$ (Partial Order MERging), which addresses the problem of merging plans for different tasks (Sec. 11.1). We first introduce a core of POPMER$_X$ merging instantaneous tasks. Then, we extend it to handle features needed by durative tasks and durative tasks with time windows.

In a summary, POPMER$_X$ is a *forward-chaining* algorithm exploiting approaches for fast heuristic-based search. Additionally, it utilises the same representation as temporally-lifted planners, which is that a durative action is split into two instances. Therefore, our algorithm handles the notion of *required concurrency*. Furthermore, we build our algorithm on partial-order planning (POP), because:

1. POP naturally structures a plan into independent plan segments and preserves the knowledge about dependencies by causal links and orderings;

2. POP follows the least-commitment approach that yields more options for merging plan segments from different plans, when compared to a totally ordered plan;

3. POP can capture the *required concurrency* in temporal planning;

12.1 POPMER$_X$

POPMER$_X$ can be visualised as several interacting components, as shown in Fig. 12.1. In the first optional layer, focused on *preprocessing*, a *batch* of input tasks is given without associated plans. Although they are *separable*, there may be certain synergies and demands, which make them *dependent* (Def. 10.5.6). This added dependency can hinder the performance of our proposed algorithm. Hence, we introduce this *preprocessing* step, based on a *relaxation* of the tasks' planning problems in order to limit their dependencies. This means that each input task $\omega_i = \langle D_i, A_i, (I, G_i), \emptyset \rangle$ is relaxed to $\omega_i^\times = \langle D_i, A_i, (I^\times, G_i), \emptyset \rangle$ by modifying the initial state. We elaborate on details in Section 12.4. Importantly, this preprocessing step can be omitted as it is not critical for the algorithm, though its presence improves the quality of the found solution.

The second layer – a wrapped planner X – then finds a plan $\pi_i^\times$ for each relaxed task. Note that the index X in the name of our algorithm represents the planner, for example POPMER$_{\text{VHPOP}}$ indicates that the VHPOP planner is wrapped within POPMER.

These relaxed plans, alongside the original initial state I and the set of the goals $\mathcal{G} = \bigcup_{i \in 1 \ldots n} G_i$, are then supplied into the third layer. This layer represents the core algorithm. In contrast to the previous two layers, which were immediate, the third layer iterates many times until a solution plan is found due to an internal feedback loop. In more detail, this third layer has the following five modules.

- The first module *reasons* about the causal structure of the input relaxed plans and chooses action points from them that are not *dependent* on any causal link or ordering. These are saved in an unsorted candidate set $\mathcal{A}_{\text{H}}^{can}$. Additionally, this module also *manages* the feedback from the fourth module – *merge* – according to which actions, links, and orderings are already *satisfied* in the solution plan candidate π_s.

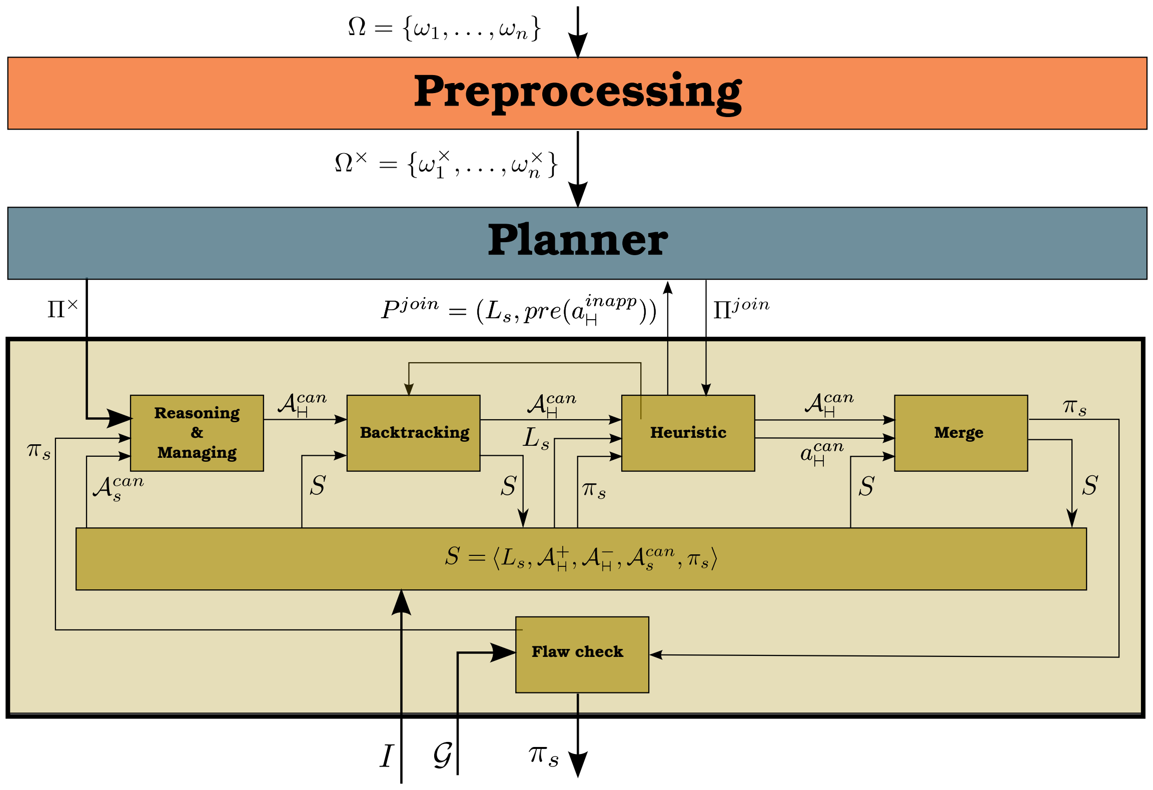


Figure 12.1: The proposed algorithm consists of three layers: optional preprocessing, an integrated state-of-the-art planner and the core of the algorithm at the bottom where merging of partially ordered plans is done. See Sec. 12.1 for detailed description.

- The candidate set $\mathcal{A}_{\mathsf{H}}^{can}$ is passed into the *backtracking* module. Backtracking is triggered when the candidate set is empty. This means that no action points could have been chosen from the input plans due to the candidate plan π_s. Therefore, the previous search state is recovered and the previously made choice is withdrawn from the recovered $\mathcal{A}_{s}^{can}$. Thus, while not optimal, POPMER$_\mathsf{X}$ is guaranteed to find a solution for the merging, if one exists.

- The candidate set $\mathcal{A}_{\mathsf{H}}^{can}$ (original or recovered) then proceeds to the *heuristic* component. Furthermore, the set of literals L_s is also part of the input to this component. The set L_s describes the current state of the world which would hold if the candidate plan would be applied to the initial state I. The heuristic module evaluates what is the best action point to be added into the candidate plan. It utilises a greedy selection mechanism based on the estimated cost of the actions. We propose different heuristics to estimate the cost, which are discussed in the corresponding section.

- The best candidate action point a_{H}^{can} is used to expand the search state S in the *merge* module.

- The module *flaw check* inspects the current plan π_s and when there are no more flaws, the algorithm ends and returns the last candidate plan. If there are still flaws, the *reasoning and managing* module choose new candidate actions and this layer continues to process the input plans as long as the candidate plan has a flaw or no solution can be found.

12.1.1 Search State

Our proposed algorithm utilises a search state, which is updated during the algorithm's iterations. In the case, when instantaneous tasks are handled, the search state is defined as:

$$S = \langle L_s, \mathcal{A}^+_{\mathsf{H}}, \mathcal{A}^-_{\mathsf{H}}, \mathcal{A}^{can}_s, \pi_s \rangle \tag{12.1}$$

where:

- $\pi_s = \langle \mathcal{A}_{\pi_s}, \mathcal{L}_{\pi_s}, \mathcal{O}_{\pi_s} \rangle$ is the candidate partial order plan that reaches the current state.
- $L_s = \{F^+, F^-\}$ is the set of positive and negative literals, respectively, that hold in S as a result of applying the candidate plan π_s to the initial state I.
- A^+_{H} is the set of *achievers* of all atoms that hold in the current state.
- A^-_{H} is the set of *deleters* of all atoms that do not hold in the current state.
- $\mathcal{A}^{can}_s$ is the set of all candidate action points that could be merged into plan π_s in the current iteration;

In the case, when durative tasks are handled, the search state is extended into:

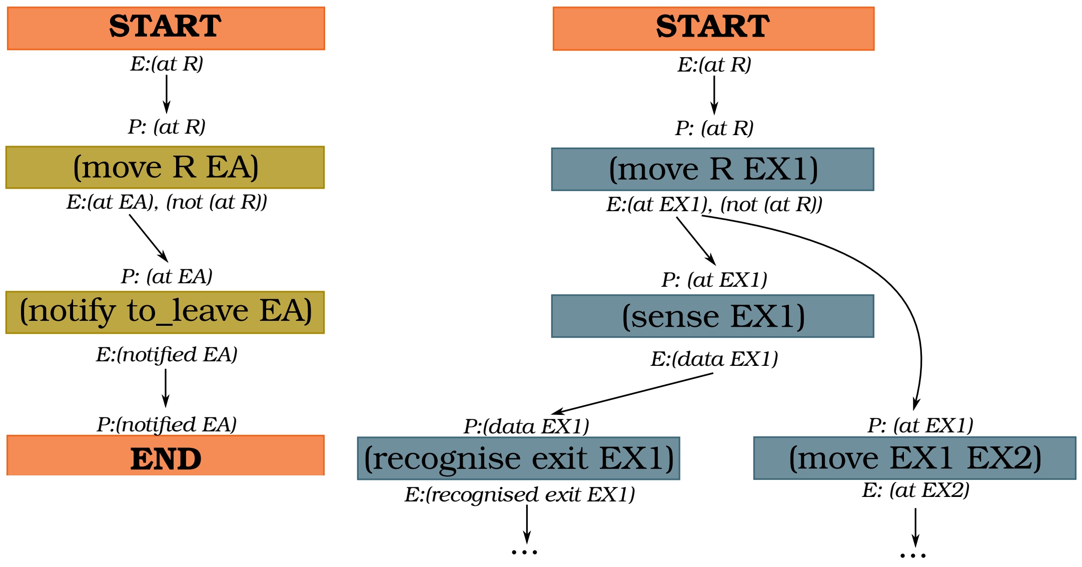
$$S = \langle t_s, L_s, \mathcal{A}^+_{\mathsf{H}}, \mathcal{A}^-_{\mathsf{H}}, \mathcal{A}^{can}_s, L^{prot}, \pi_s \rangle \tag{12.2}$$

where

- $t_s = \text{makespan}(\mathcal{O}_{\pi_s})$ represents the minimal duration (i.e. makespan) of the candidate plan π_s. This time indicator is used to in the heuristic module.
- L^{prot} is the set of *protected* literals by the current state.

12.2 Example

In order to illustrate the functionality of POPMER$_{\mathsf{X}}$, we extend the example given in Sec. 8.1. Alongside this example, we informally introduce a number of



(a) π_1 (b) π_2

Figure 12.2: Details of partially ordered plans for task ω_1 (a) and ω_2 (b). The arrows visualise the causal links between actions. The dots represent that there are more actions in the plan.

terms that are exploited in the algorithm. Their formal definitions are given in the next section.

The two instantaneous tasks ω_1 and ω_2 from the example scenario (Sec. 1.1) are given to POPMER$_\mathrm{X}$. Preprocessing is not considered in this example. Therefore, a planner X returns two partially ordered plans $\pi_1 = \langle \mathcal{A}_1, \mathcal{L}_1, \mathcal{O}_1 \rangle$ and $\pi_2 = \langle \mathcal{A}_2, \mathcal{L}_2, \mathcal{O}_2 \rangle$, which are visualised in Fig. 12.2a and Fig. 12.2b, respectively. We refer to actions by a subscript $x - y$, where $x \in \{1, 2\}$ refers to either plan π_1 or π_2 and y refers to the order of the action within the plan. Hence, action $a_{\mathrm{H}1-1}$ is the first action in the plan π_1. Causal links are denoted similarly by a subscript $x - y - z$. Whilst x has the same meaning, y and z refer to the actions between which the causal link exists. For example, causal link $\ell_{2-1-2} : a_{\mathrm{H}2-1} \xrightarrow{\text{(at EX1)}} a_{\mathrm{H}2-2}$ is between the first and the second action from plan π_2. Finally, the ordering subscript follows the same pattern as causal links.

First, the search state S is initialised as:

$$S = \langle \{\text{(at R)}\}, \{a_{\mathsf{H}0} : \text{(at R)}\}, \emptyset, \emptyset, \langle \{a_{\mathsf{H}0}\}, \emptyset, \emptyset \rangle \rangle,$$

where the candidate plan π_s is initialised by the START action $a_{\mathsf{H}0}$, whose effects reflect the initial state $I =$ (at R). Hence, if the plan π_s was executed, the current state of the environment would be $L_s = \{\text{(at R)}\}$. Moreover, the START action *achieves* the literal (at R); there is no deleter. Then, the iteration between the five modules is triggered.

The *reasoning and managing* module extracts actions from the provided plans that can be merged. In the first iteration, either $a_{\mathsf{H}2-1} =$ (move R EX1) or $a_{\mathsf{H}1-1} =$ (move R EA) can be chosen because they are only *dependent* on the START action, which is already merged. The START action is a *provider* for $a_{\mathsf{H}1-1}$ and $a_{\mathsf{H}2-1}$. Other actions cannot be chosen yet as their *providers* are not merged. As a result, the candidate set is $\mathcal{A}_{\mathsf{H}}^{can} = \{a_{\mathsf{H}1-1}, a_{\mathsf{H}2-1}\}$.

This candidate set is not empty, therefore backtracking is not triggered. The candidate set is passed into the *heuristic* module. We provide more details about the proposed heuristic later. Let us now assume that the module chooses $a_{\mathsf{H}2-1} =$ (move R EX1) as the candidate a_{H}^{can}. Therefore, this action is *merged* into the plan π_s. Merging not only adds the action into the set of actions $\mathcal{A}_{\pi f}$ but also orders it after the START action and establishes causal links between the effects of the START action and preconditions of the chosen action. We add both causal link and ordering because they reflect on different reasons why such constraints need to be added between the actions. However, they have the same effect on the final order of actions in the candidate plan in this case. Hence, the merged plan is:

$$\pi_s = \langle \{a_{\mathsf{H}0}, a_{\mathsf{H}2-1}\}, \{a_{\mathsf{H}0} \xrightarrow{\text{(at R)}} a_{\mathsf{H}2-1}\}, \{a_{\mathsf{H}0} < a_{\mathsf{H}2-1}\} \rangle.$$

The state S is updated as follows:

$$\begin{aligned}
L_s =& \{\text{(at EX1), (not (at R))}\}, \\
\mathcal{A}_{\mathsf{H}}^{+} =& \{a_{\mathsf{H}2-1} : \text{(at EX1)}\}, \\
\mathcal{A}_{\mathsf{H}}^{-} =& \{a_{\mathsf{H}2-1} : \text{(at R)}\}, \\
\mathcal{A}_{s}^{can} =& \{a_{\mathsf{H}1-1}, a_{\mathsf{H}2-1}\}, \\
\pi_s =& \text{ as defined above}
\end{aligned}$$

Notice that achievers and deleters in the state S represent only the atoms holding in the current L_s; this implies from the closed world assumption. Therefore, the previous achiever $a_{\mathsf{H}0}$: (at R) is not presented anymore.

In the next iteration, the *reasoning and managing* module chooses new candidate actions. It receives the old candidate set $\{a_{\mathsf{H}1-1}, a_{\mathsf{H}2-1}\}$ as its input. First, it removes the previously merged action $a_{\mathsf{H}2-1}$ and the action $a_{\mathsf{H}1-1}$ = (move R EA) remains in the candidate set. Moreover, action $a_{\mathsf{H}2-1}$ from the plan π_2 has become *satisfied* in π_s. Also, the causal link $\ell_{2-0-1} : \{a_{\mathsf{H}0} \xrightarrow{\text{(at R)}} a_{\mathsf{H}2-1}\}$ and the ordering $o_{2-0-1} : a_{\mathsf{H}0} < a_{\mathsf{H}2-1}$ has become *satisfied* in π_s. Therefore, the next action $a_{\mathsf{H}2-2}$ = (sense EX1) can be chosen as a candidate. As a result, $\mathcal{A}_{\mathsf{H}}^{can} = \{a_{\mathsf{H}1-1}, a_{\mathsf{H}2-2}\}$.

The heuristic module recognises that action $a_{\mathsf{H}1-1}$ is *inapplicable* (Def. 10.3.3) in the current state L_s because its precondition requires that literal (at R) holds, but the current state contains its negation. Moreover, the causal link $\ell_{1-0-1} : a_{\mathsf{H}0} \xrightarrow{\text{(at R)}} a_{\mathsf{H}1-1}$ is *violated* by the current state L_s. As a result, the heuristic module cannot compute the heuristic value for action $a_{\mathsf{H}1-1}$. Therefore, the heuristic module must first obtain a *joining plan*.

Joining plans are obtained only for those actions that are inapplicable in L_s in order to measure the cost of adding such an action into the candidate plan. Hence, for each inapplicable action point a_{H}^{inapp}, a new problem $P^{join} = (L_s, pre(a_{\mathsf{H}}^{inapp})$ is passed into the planner X to receive the joining plan π^{join}. The joining plan for action $a_{\mathsf{H}1-1}$ = (move R EA) contains only action (move EX1 R) in this case. Finally, the heuristic module chooses action $a_{\mathsf{H}2-2}$ = (sense EX1) to be merged as its cost is smaller than the cost of the joining plan. This results in the new merged plan:

$$\pi_s = \langle \{a_{\mathsf{H}0}, a_{\mathsf{H}2-1}, a_{\mathsf{H}2-2}\}, \{a_{\mathsf{H}0} \xrightarrow{\text{(at R)}} a_{\mathsf{H}2-1}, a_{\mathsf{H}2-1} \xrightarrow{\text{(at EX1)}} a_{\mathsf{H}2-2}\},$$
$$\{a_{\mathsf{H}0} < a_{\mathsf{H}2-1}, a_{\mathsf{H}0} < a_{\mathsf{H}2-2}, a_{\mathsf{H}2-1} < a_{\mathsf{H}2-2}, \}\rangle.$$

Additionally, the actions $a_{\mathsf{H}2-1}, a_{\mathsf{H}2-2}$ represent a *plan segment*, i.e. a continuous subset of actions from a single plan.

The algorithm continues iterating while the candidate plan π_s has a flaw – an unsatisfied goal G. The solution plan cannot contain a goal related flaw. Therefore, the merged final plan π_f achieves all tasks goals but they do not have to hold at the end of the plan. As a result, we will define when *a goal G_i is satisfied*.

12.3 Terminology Related to Plan Merging

We formalise here the terminology introduced in the previous example. Whilst the example considers instantaneous tasks, the used terms also hold for temporal plans.

12.3.1 Extending POP definition

Causal links and *orderings* in a POP cause a *dependency* between action points. Therefore, they can be united as *dependants* and *providers*.

Definition 12.3.1. ***Dependant:*** *A dependant is an action point a'_{H} such that there exists either a causal link $a_{\mathsf{H}} \xrightarrow{l} a'_{\mathsf{H}}$ or there is ordering $a_{\mathsf{H}}\, o\, a'_{\mathsf{H}}$. Hence, a'_{H} is* dependent on *action point a_{H}. Importantly, an action's end point is always dependent on its start point.*

Definition 12.3.2. ***Provider:*** *A provider is an action point a_{H} such that one or more other action points are dependent on it. Note that an action start point has always at least one dependant, its end point. We define the set $\mathcal{A}^{dep}$ of all providers for action point a_{H} as $\mathcal{A}^{dep}_{a_{\mathsf{H}}} = \{a'_{\mathsf{H}} \in \mathcal{A}_{\mathsf{H}} \mid (a'_{\mathsf{H}} \xrightarrow{l} a_{\mathsf{H}}) \vee (a'_{\mathsf{H}} \, o \, a_{\mathsf{H}})\}$.*

Furthermore, given two partially ordered plans $\pi_1 = \langle \mathcal{A}_1, \mathcal{L}_1, \mathcal{O}_1 \rangle$ and $\pi_2 = \langle \mathcal{A}_2, \mathcal{L}_2, \mathcal{O}_2 \rangle$, their actions, causal links and orderings can be compared.

Definition 12.3.3. ***Satisfied action in a plan:*** *An action $a_i \in \mathcal{A}_1$ is* satisfied in plan *π_2 if and only if $a_i \in \mathcal{A}_2$.*

Definition 12.3.4. ***Satisfied causal link in a plan:*** *A causal link $\ell_{ij} : a_i \xrightarrow{l} a_j,\ \ell_{ij} \in \mathcal{L}_1$ is* satisfied in plan *π_2 if and only if $\ell_{ij} \in \mathcal{L}_2$, which implies that $a_i,\ a_j$ are satisfied in π_2.*

Definition 12.3.5. ***Satisfied ordering in a plan:*** *An ordering $o_{ij} : a_i \, o \, a_j,\ o_{ij} \in \mathcal{O}_1$ is* satisfied in plan *π_2 if and only if $o_{ij} \in \mathcal{O}_2$, , which implies that $a_i,\ a_j$ are satisfied in π_2.*

A POP plan π can be also extended by an action point.

Definition 12.3.6. ***Plan expansion operator $\oplus$:*** *Given $\pi = \langle \mathcal{A}, \mathcal{L}, \mathcal{O} \rangle$ and action point a_{H}, the operator $\pi \oplus a_{\mathsf{H}}$ has the following effects. First, all action points $\mathcal{A}$ are ordered before the new point a_{H}, hence $\mathcal{O} = \mathcal{O} \cup \{\langle a'_{\mathsf{H}} < a_{\mathsf{H}} \rangle \mid a'_{\mathsf{H}} \in \mathcal{A}\}$. Then, the action point a_{H} is added to the plan action points $\mathcal{A} = \mathcal{A} \cup a_{\mathsf{H}}$.*

12.3.2 Joining Plans

A joining plan is needed if an *inapplicable* action point a_{H} is supposed to expand the current state described by L_s. Thus, not all of its preconditions are satisfied in this state, i.e., $\exists\, l \in pre(a_{\mathsf{H}}) : l \notin L_s$. Such a situation occurs because a causal link where a_{H} is the *consumer* (Def. 10.4.5) was *violated*.

Definition 12.3.7. ***Violated causal link:*** *Causal link $\ell = a'_{\mathsf{H}} \xrightarrow{l} a_{\mathsf{H}}$ becomes* violated *if (i) the producing action point a'_{H} has been already merged into the plan but the consuming* *action point a_{H} has not; and (ii) another merged action from a different plan has caused that literal l (required by the causal link) does not hold in the current state, i.e. $l \notin L_s$.*

For such violated link, a joining plan needs to be obtained in order to satisfy literal l in the current state. This corresponds to the fact that a_{H} is inapplicable action point in the current state.

Definition 12.3.8. ***Joining plan:*** *A joining plan for an action point a_{H} first obtains a temporary plan that satisfies the action point's preconditions, i.e., $\pi^{temp} = X(D_i, A_i, (L_s, pre(a_{\mathsf{H}}))$ where D_i, A_i stands for the domain and the action set of the task from where the action point comes from. Then, the* joining plan *is obtained by adding the action point to the temporary plan, i.e., $\pi^{join} = \pi^{join} \oplus a_{\vdash}$.*

12.3.2.1. Unsolvable Temporary Problems

Importantly, the generated temporary problem $(L_s, pre(a_{\mathsf{H}}))$ may be unsolvable in the given domain D_i. This can be illustrated again on the example from Sec. 12.2. We have left the example after the second iteration when the candidate plan was containing the actions (START), (move R EX1), (sense EX1). For the rest of this explanation, we are going to work with *durative* actions. Assume that the durative action (move ?from ?to) is modelled as follows:

- preconditions: (at start (at ?from));
- effects: (at start (not (at ?from))) and (at end (at ?to)).

We skip forward in the iterations to the search state, where $L_s = \{$ (data EX1), (recognised exit EX1), (at R) $\}$. In this state there are two candidates: $a_{\vdash 1-1}$ = (move R EA) or $a_{\vdash 2-4}$ = (move EX1 EX2). Note that both are start action points. The heuristic chooses $a_{\vdash 1-1}$. The merging module merges this action point and applies its effect (not (at R)) to L_s. As a result, the search state is updated to $L_s = \{$ (data

EX1), (recognised exit EX1)}. Note that this state does not include any literal describing where the robot is.

In the next iteration, the candidates are: $a_{\vdash 2-4}$, $a_{\dashv 1-1}$. Because $a_{\vdash 2-4}$ is inapplicable in the current state, the heuristic module needs to obtain a joining plan. Therefore, the planning problem is created with an initial state $I = L_s = \{$ (data EX1), (recognised exit EX1)$\}$ and the goal $G =$ (at EX1). However, a plan cannot be found for this problem because the initial state does not contain literal (at ?loc).

To conclude, this is a problem of domain modelling. However, exactly this situation occurs in the Driverlog domain from the International Planning Competition (IPC). Moreover, we have notice that some off-the-shelf planners can recognise invalid planning problem very fast, but some not. Therefore, POPMER$_{\text{X}}$ needs to handle the situation when its wrapped planner X keeps searching for a solution to an invalid problem. Currently, we provide a runtime threshold to stop the planner X and claim that the problem is unsolvable. We provide more discussion of this phenomena in the evaluation.

12.3.3 Plan Segment

A plan π can be separated into several *plan segments*. A plan segment is any continuous subset of action points from π.

Definition 12.3.9. ***Plan segment:*** *A plan segment is an ordered set of action points, containing one or more action points from a plan π. If a'_{H} is the start of the plan segment and a_{H} is the end, all action points from π ordered between a'_{H} and a_{H} must be part of the plan segment.*

A plan segment can contain a single action point or its last element is a start action point as in the case of a joining plan. Because our heuristic module needs to estimate the cost of this joining plan, which is based on the duration, we need to define the duration of a plan segment. Our motivation is to measure how much time the temporary plan and the *whole* inapplicable action are going to consume. Therefore, the duration of the plan segment is defined as follows.

Definition 12.3.10. ***Duration of plan segment:*** *The duration of a plan segment is expressed via its first element $a'_{\mathsf{H}} = (a', t'_{\mathsf{H}})$ and the latest action end point $a_{\dashv} = (a, t_{\dashv})$. This end point does not have to be in the plan segment but its corresponding start point $a_{\vdash}$ must be included. Hence, the duration of plan segment, $d = t_{\dashv} - t'_{\mathsf{H}}$, expresses how much time is needed for all actions that have started within the plan segment to finish.*

12.3.4 Satisfied Goals

The merged plan *satisfies* tasks' goals.

Definition 12.3.11. ***Satisfied goal:*** *Goal G_i of instantaneous or durative task ω_i is satisfied in plan π_f if goal literals L_G are satisfied in some state S visited by the plan, i.e. $L_G \subset L_s$.*

Definition 12.3.12. ***Satisfied goal with time constraints:*** *Goal G_i of durative task ω_i with time constraints is satisfied in plan π_f if goal literals L_G are satisfied in some state S visited by the plan and its minimal duration t_s is between the release date and deadline of the task, hence $r_i \leq t_s \leq d_i$.*

12.3.4.1. Flaws in Merged Plans

Definition 12.3.13. ***Goal related flaw:*** *The merged plan π_s has as many goal related flaws γ_G as there are unsatisfied goals. All goal related flaws need to be solved in the solution plan.*

Definition 12.3.14. ***Action related flaw:*** *The merged plan π_s has as many action related flaws γ_A as there are unsatisfied actions.*

We allow the solution plan to have action related flaws. This may be atypical and one might wonder why we accept these flaws. For example, if two plans contain the same action, only one would be in the solution plan. Therefore, the unmerged action is an action related flaw, but it does not violate the solution plan. However, we need the notion of action related flaws because goal flaws cannot be eliminated directly. They are removed by merging actions from the original input plans, i.e., by reducing action related flaws.

12.4 Preprocessing via Problem Relaxation

Demands create dependencies between plan segments from different tasks. The problem is that if plans are obtained for each task separately, then demands are optimised locally. While merging these plans into the single final plan, the algorithm must obey the locally optimised input plans. Such limitations often lead to a lower plan quality, in comparison to the conjoining approach. This can be illustrated again on the example from Sec. 12.2.

12.4.1 Example

We have left the example after the second iteration when the candidate plan contained the actions (START), (move R EX1), (sense EX1). In the third iteration, the (recognise exit EX1) is to be merged. In the fourth iteration, there are two candidates: $a_{\mathsf{H}1-1}$ = (move R EA) or $a_{\mathsf{H}2-4}$ = (move EX1 EX2). Because the action $a_{\mathsf{H}1-1}$ is *inapplicable* in the current state, the *heuristic* module obtains a joining plan. In this case, the joining plan consists only of action (move EX1 R) and (move R EA). The cost of this joining plan is smaller than the cost of action $a_{\mathsf{H}2-4}$. Therefore, the heuristic planner updates the candidate list by the action (move EX1 R), which is chosen to be merged in this iteration. Finally, in the fifth iteration, action (move R EA) is chosen.

The pair of actions (move EX1 R), (move R EA) prolongs the makespans. A better solution would be to include action (move EX1 EA), which is shorter that the pair. However, the locally optimised demands in the input plans do not allow that. If those demands did not appear in the input plans, the merging algorithm would have more freedom in how to merge the plans, leading to a shorter makespan. To conclude, the preprocessing module addresses how the local optimisation of demands in the single plans can be limited in order to provide more independence on those decisions in the merging algorithm.

12.4.2 Elimination of Demands

The elimination of demands is based on *plan relaxation*, hence the initial state I is extended by all literals that satisfy the demands leading to the relaxed initial state $I^{\times}$ and relaxed tasks such as $\omega_i^{\times} = \langle D_i, A_i, (I^{\times}, G_i), \pi_i^{\times} \rangle$. For example, the spatial constraints from the above example can be avoided by adding literals that the robot is everywhere in the environment, i.e., the initial state contains (at R), (at EX1), (at EX2), (at EA). After such relaxation, the resulting plans do not contain any actions restricting the order between plan segments from different tasks, in this case (move ?from ?to) action. Hence, the input plans are not affected by the demands and the merging algorithm can optimise them in the overall plan. Currently, this relaxation is domain specific.

12.4.3 Automatic Relaxation

Literals that need to be relaxed can be extracted from a planning problem by a POP planner. The planner would plan for a conjunction of all tasks' goal literals. If the tasks were independent, their plans would represent parallel branches in the POP structure. However, if there is a dependency between them, the POP planner would detect a threat imposed by effect literals of an action from one task to a causal link from another task. For example, Fig. 12.3 illustrates such a situation

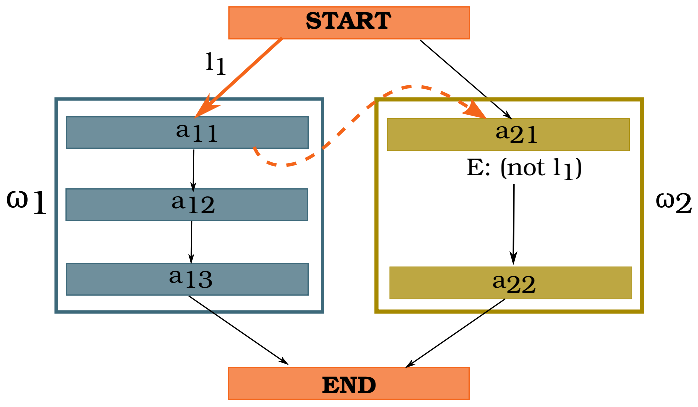


Figure 12.3: The effect literal (not l_1) of action a_{21} belonging to the plan of task ω_2 imposes a threat on the causal link $START \xrightarrow{l_1} a_{11}$ belonging to the task ω_1. Therefore, the POP planner would need to add an ordering (marked by dashed line) to solve this threat. This ordering represents the dependency between both tasks. In order to prevent this dependency, literal l_1 would be relaxed.

where the effect literal (not l_1) of action a_{21} belonging to the plan of task ω_2 imposes a threat on the causal link $START \xrightarrow{l_1} a_{11}$ belonging to the task ω_1. Hence, the POP planner would need to add an ordering (marked by dashed line) to solve this threat. This ordering represents the dependency between both tasks. In order to prevent this dependency, literal l_1 would be added to the relaxed initial state and all the process repeated until there are no ordering needed between actions from different tasks.

This approach has a drawback - we perform the conjoining planning problem in order to find the literals to relax. However, we do not need to include all tasks instances. It is sufficient to plan only for two instances of each task's type. For example, in our scenario, there are only two types of tasks which an user can request: either check emergency exits or notify people at some location. Therefore, the aforementioned analysis would need to plan for four tasks: two randomise instances for each type of a task. We have not included this automatic relaxation into our thesis as it would require to have control over the POP planner. As we intend to use a POP planner as a black box in this thesis, this step would require significant effort in modifying an existing POP planner or writing one from a scratch.

Therefore, we leave addressing the development of an algorithm that detects demands, and provides for their relaxation automatically for future work.

12.5 Planner

Any planner that is able to parse PDDL 2.1 can be wrapped in our algorithm. Currently, POPMER supports only such planners that provide standardised output required by the VAL validator[1]. For other planners, a parser of their plans would need to be implemented. The planner's properties will affect our algorithm. For example, if an incomplete algorithm is used, our algorithm will become incomplete as well.

12.6 Plan Merging with Instantaneous Tasks

We describe the core of our algorithm in three stages. In this section, we focus on the simplified problem with instantaneous tasks in order to formally describe the core of our proposed merging algorithm POPMER$_{\mathsf{X}}$, Hence, the algorithm operates on instantaneous actions that are mapped into instantaneous action points $\mathcal{A}_{\perp}$, which are equal to general action points $\mathcal{A}_{\mathsf{H}}$, in this case. Then, we add features supporting temporal planning for durative tasks in the following section. Finally, we extend the algorithm in order to handle time windows (Sec. 12.8).

12.6.1 Reasoning and Managing

This module manages the relaxed input plans $\pi_i^{\times}$ and reasons with action points in order to extract those that can be merged into π_s, see Algorithm 4. The input to this module is a set $\Pi^{\times} = \{\pi_1^{\times}, \ldots, \pi_n^{\times}\}$ of relaxed plans, the current can-

[1]https://github.com/KCL-Planning/VAL

didate plan π_s and the set of the previous candidate action points A^{can}_{H}. First, the algorithm removes the lastly merged action point from the candidate set. Then, it adds new candidate actions points from the input plans. For each candidate $a^{can}_{\mathsf{H}} \in \mathcal{A}^{can}_{\mathsf{H}}$, the following two conditions must hold:

- a^{can}_{H} has not been merged before;
- all *providers*, i.e. other action points on which the candidate depends on, are merged. Recall the set $\mathcal{A}^{dep}_{a_{\mathsf{H}}}$ that contains all providers for a_{H}.

The unmerged and merged action points to reason on $\mathcal{A}^{unm}_{\mathsf{H}}, \mathcal{A}^{m}_{\mathsf{H}}$, respectively, are obtained by the method *merged-action-points($\pi^{\times}_i$, π_s)*. This method analyses which of the action points from the input plan are *satisfied* in the current plan π_s. Therefore, it splits the action points into $\mathcal{A}^{\times}_{\mathsf{H}i} = \mathcal{A}^{unm}_{\mathsf{H}} \cup \mathcal{A}^{m}_{\mathsf{H}}$. Then, the candidate set $\mathcal{A}^{can}_{\mathsf{H}}$ can be updated by such action points that satisfy the aforementioned conditions, see Line 5.

Algorithm 4: extract
```
Input: $\Pi^{\times}$, $\pi_s$, $\mathcal{A}^{can}_{\mathsf{H}}$
Output: $\mathcal{A}^{can}_{\mathsf{H}}$
1: $a^{last}_{\mathsf{H}}$ = get-last-action-point($\pi_s$)
2: $\mathcal{A}^{can}_{\mathsf{H}} = \mathcal{A}^{can}_{\mathsf{H}} \setminus \{a^{last}_{\mathsf{H}}\}$
3: for $\pi^{\times}_i = \langle \mathcal{A}^{\times}_i, \mathcal{L}^{\times}_i, \mathcal{O}^{\times}_i \rangle \in \Pi^{\times}$ do
4:     $\mathcal{A}^{unm}_{\mathsf{H}}, \mathcal{A}^{m}_{\mathsf{H}}$ =merged-action-points($\pi^{\times}_i$, $\pi_s$)
5:     $\mathcal{A}^{can}_{\mathsf{H}} = \mathcal{A}^{can}_{\mathsf{H}} \cup \{a_{\mathsf{H}} \in \mathcal{A}^{unm}_{\mathsf{H}} \mid a_{\mathsf{H}} \notin \mathcal{A}^{can}_{\mathsf{H}} \wedge (\forall a'_{\mathsf{H}} \in \mathcal{A}^{dep}_{a_{\mathsf{H}}} : a'_{\mathsf{H}} \in \mathcal{A}^{m}_{\mathsf{H}})\}$
6: end for
7: return $\mathcal{A}^{can}_{\mathsf{H}}$
```

12.6.2 Backtracking

If the candidate set A^{can}_{H} is empty, it signals that the current candidate plan π_s does not allow merging of a new action point due to dependency issues. Hence, the backtracking module recovers the previous search state and propagates the previous choice as invalid, see Alg. 5.

Algorithm 5: backtrack

```
Input: $\mathcal{A}_{\mathsf{H}}^{can}$, $S$
Output: $\mathcal{A}_{\mathsf{H}}^{can}$, $S$
1: if $\mathcal{A}_{\mathsf{H}}^{can} = \emptyset$ then
2:     $a_{\mathsf{H}}^{last}$ = get-last-action-point($\pi_s$)
3:     $S =$ parent-state($S$)
4:     $\mathcal{A}_{s}^{can} = \mathcal{A}_{s}^{can} \setminus a_{\mathsf{H}}^{last}$
5:     $\mathcal{A}_{\mathsf{H}}^{can} = \mathcal{A}_{s}^{can}$
6: end if
7: return $\mathcal{A}_{\mathsf{H}}^{can}, S$
```

First, the method *get-last-action-point* returns the action point that was added in the last iteration. Then, the previous state S is recovered and the last action is removed from the set of possible candidates. Note that this action will be added again to the candidates in some future iteration.

12.6.3 Heuristic Candidate Selection

This module analyses the candidates $\mathcal{A}_{\mathsf{H}}^{can}$ according to the set of literals L_s, updates the candidate list (if necessary) and, finally, chooses the best candidate action, see Algorithm 6. It begins by separating the candidate set to those action points that are *applicable* in the current state L_s, see Def. 10.3.3, and to those who are not, i.e., $\mathcal{A}_{\mathsf{H}}^{can} = \mathcal{A}_{\mathsf{H}}^{app} \cup \mathcal{A}_{\mathsf{H}}^{inapp}$. Any action point from the applicable set $\mathcal{A}_{\mathsf{H}}^{app}$ can be passed into the *merge* module but for any action in the inapplicable set $\mathcal{A}_{\mathsf{H}}^{inapp}$, a *joining* plan, see Def. 12.3.8, needs to be found first. To give an informal reminder of the definition, the joining plan reaches a state where all preconditions of inapplicable action point are satisfied. Moreover, such a plan is extended by the inapplicable action point by the operator $\oplus$. As a result, this module acquires a set of joining plans Π^{join} for all the inapplicable actions.

If there are no applicable actions and no joining plans can be found, then an empty set is returned in order to trigger backtracking again, as the current state cannot be expanded. Otherwise, a *heuristic* function not only decides what is the

Algorithm 6: heuristically-choose

```
Input: $\mathcal{A}_{\mathsf{H}}^{can}, L_s, \pi_s$
Output: $a_{\mathsf{H}}^{can}$, $\mathcal{A}_{\mathsf{H}}^{can}$
1: $\mathcal{A}_{\mathsf{H}}^{app} = \{a_{\mathsf{H}} \in \mathcal{A}_{\mathsf{H}}^{can} \mid pre(a_{\mathsf{H}}) \subseteq L_s\}$
2: $\mathcal{A}_{\mathsf{H}}^{inapp} = \mathcal{A}_{\mathsf{H}}^{can} \setminus \mathcal{A}_{\mathsf{H}}^{app}$
3: $\Pi^{join} = \{X\,(D_i, A_i, (L_s, pre(a_{\mathsf{H}}))) \oplus a_{\mathsf{H}} \mid a_{\mathsf{H}} \in \mathcal{A}_{\mathsf{H}}^{inapp}\}$
4: if $\mathcal{A}_{\mathsf{H}}^{app} = \emptyset$ and $\Pi^{join} = \emptyset$ then
5:     return $\emptyset$, $\mathcal{A}_{\mathsf{H}}^{can}$
6: else
7:     $a_{\mathsf{H}}^{can}$, $\mathcal{A}_{\mathsf{H}}^{can}$ = heuristic($\mathcal{A}_{\mathsf{H}}^{app}, \Pi^{join}, \pi_s$)
8:     return $a_{\mathsf{H}}^{can}$, $\mathcal{A}_{\mathsf{H}}^{can}$
9: end if
```

best candidate action point to be merged but also updates the candidate set by action points from the chosen plan segment (see next section for further details).

12.6.3.1. Heuristic and Greedy Selection Mechanism

The heuristic function first heuristically chooses the best *plan segment* π^{can} from $\mathcal{A}_{\mathsf{H}}^{app}$ or Π^{join}. Then, it utilises the reasoning and managing module in order to extract only one action point that can be merged into π_s. Notice that there is always at least one action point that will be chosen; either, it is an action point from $\mathcal{A}_{\mathsf{H}}^{app}$ or it is the first action point in the best joining plan from Π^{join}. If there are more action points extracted in the above step, such as action points in the joining plan that can be in parallel, the following *rules* choose the action point:

- whose action has the shorter duration (this rule makes no difference in instantaneous planning but it is important in temporal planning);
- with fewer negative literals in its effects;
- without *over all* preconditions (again only for temporal planning);
- with fewer preconditions;

The rules are applied in the order, for example, if the first rule cannot decide, the second is applied, etc. Finally, more details are in Algorithm 7.

Algorithm 7: heuristic

```
Input: $\mathcal{A}_{\mathsf{H}}^{app}, \Pi^{join}, \pi_s$
Output: $a_{\mathsf{H}}^{can}$, $\mathcal{A}_{\mathsf{H}}^{can}$
1: $\pi^{can} = h^{link}(\mathcal{A}_{\mathsf{H}}^{app}, \Pi^{join})$
2: $\mathcal{A}_{\mathsf{H}}^{can} =$ extract($\{\pi^{can}\}, \pi_s, \mathcal{A}_{\mathsf{H}}^{can}$)
3: $a_{\mathsf{H}}^{can} =$ rules($\mathcal{A}_{\mathsf{H}}^{can}$)
4: return $a_{\mathsf{H}}^{can}$, $\mathcal{A}_{\mathsf{H}}^{can}$
```

As we focus on instantaneous planning in this chapter, the makespan of the plan is defined by the number of actions in the plan. Hence, the proposed heuristic h^{link} minimises the makespan by limiting the occurrence of joining plans. This is achieved by considering the causal links of the input plans:

h^{link} : choose the *plan segment* that does not violate a causal link $a_i^m \xrightarrow{l} a_i^{unm}$ in any input plan $\pi_i^{\times}$ such that the *producing* action is already merged but the the *consumer* is not.

This gives the heuristic a global look ahead, this means that the heuristic is guided which literals in the current state L_s need to be preserved to the future. We provide more discussion about heuristic in the next chapter as temporal planning adds to its difficulty.

12.6.4 Merge Algorithm

The merge algorithm (Algorithm 8) expands the current state S by merging the best candidate action point a_{H}^{can}. The search state is updated as follows. First, the set of active literals, achievers and deleters are updated by the effects of the candidate action point. Then, the set of candidate action points applicable in this state is saved for possible backtracking. The candidate action point is finally added to the set of action point $\mathcal{A}_{\pi_s}$.

For each precondition of the action, a new causal link and corresponding ordering are added between the achiever or deleter (for negative literals) and the merged action point. As our heuristic is greedy and reasons about what is the best action to be added *at the end* of the plan π_s, we need to add ordering to ensure that this action point is ordered *after* the last action point a_{H}^{last}, see Line 8. Without this, our heuristic estimate would not be valid.

Algorithm 8: merge

```
Input: $a_{\mathsf{H}}^{can}, \mathcal{A}_{\mathsf{H}}^{can}, S$
Output: $S$
1: $L_s = L_s \cup \mathit{eff}(a_{\mathsf{H}}^{can})$
2: $\forall f \in \mathit{eff}^{+}(a_{\mathsf{H}}^{can}) :\ a_f^{+} = a_{\mathsf{H}}^{can}$
3: $\forall f \in \mathit{eff}^{-}(a_{\mathsf{H}}^{can}) :\ a_f^{-} = a_{\mathsf{H}}^{can}$
4: $\mathcal{A}_s^{can} = \mathcal{A}_{\mathsf{H}}^{can} \setminus \{a_{\mathsf{H}}^{can}\}$
5: $\mathcal{A}_{\pi_s} = \mathcal{A}_{\pi_s} \cup \{a_{\mathsf{H}}^{can}\}$
6: $\forall l \in \mathit{pre}^{+}(a_{\mathsf{H}}^{can}) : \mathcal{L}_{\pi_s} \cup \langle a_l^{+} \xrightarrow{l} a_{\mathsf{H}}^{can}\rangle$,
   $\mathcal{O} = \mathcal{O} \cup \langle a_l^{+} \prec a_{\mathsf{H}}^{can}\rangle$
7: $\forall l \in \mathit{pre}^{-}(a_{\mathsf{H}}^{can}) : \mathcal{L}_{\pi_s} \cup \langle a_l^{-} \xrightarrow{l} a_{\mathsf{H}}^{can}\rangle$,
   $\mathcal{O} = \mathcal{O} \cup \langle a_l^{-} \prec a_{\mathsf{H}}^{can}\rangle$
8: $\mathcal{O} = \mathcal{O} \cup \langle a_{\mathsf{H}}^{last} < a_{\mathsf{H}}^{can}\rangle$
9: return $S$
```

12.6.5 Flaw Check

This module inspects the current plan π_s in order to analyse if all goals $\mathcal{G} = \cup_{i\in\{1,\dots,n\}} G_i$ are satisfied (Def. 12.3.11) in the plan. If yes, it returns plan π_s as the output of POPMER$_X$. If not, a new iteration step is triggered by passing plan π_s to the *reasoning and managing* module.

12.7 Plan Merging with Durative actions

Durative actions add a few features that plan merging must cover:

- Durative action a is represented by two action points: its start action point $a_{\vdash} = (a, t_{\vdash})$ and its end action point $a_{\dashv} = (a, t_{\dashv})$, which are dependent on each other. The dependency is expressed via the temporal constraint

$t_{\dashv} = t_{\vdash} + p$, where p is the duration of action a. Hence, this constraint must be respected when merging action points, and affects the *reasoning and managing* and *heuristic* modules.

- Preconditions and effects of durative actions are related to the start or end action point. Moreover, preconditions can be also required to hold *over all* duration of the action. Such literals, $L_{\leftrightarrow}$, are denoted as *invariants*. Especially these literals complicate the merging algorithm as they must be protected from violation during the interval when the start action point has been already merged but the end action point has not. We have introduced the notion of *protected literal* (Def. 10.3.4) in order to ensure consistency of over all literals during action. Therefore, the plan merging needs to keep track of the currently protected literals. In a similar manner as before, this limits the possible candidates that can be chosen. Candidates violating protected literals are removed in the *heuristic* module.

- The *heuristic* must now consider the duration of actions in order to minimise the overall duration of the final plan.

In this chapter, we highlight the required changes to the modules. Note that the backtracking and flaw checking modules remain unchanged. Furthermore, $a_{\mathsf{H}} \in \mathcal{A}_{\mathsf{H}}$ now refers to either start action point $a_{\vdash}$ or end action point $a_{\dashv}$ where $\mathcal{A}_{\mathsf{H}} = \mathcal{A}_{\vdash} \cup \mathcal{A}_{\dashv}$.

12.7.1 Reasoning and Managing

We have introduced two conditions that must hold for an action point to be extracted as a candidate: it has not yet been merged and its *providers* are merged. In temporal planning, an end action point can only be chosen if its start action point and all its providers have already been merged. Note that the providers for the end action point are those action points which satisfy the *at end* preconditions $pre(a)_{\dashv}$. However, as we have defined that an end action point is always dependent on its start action point, see Def. 12.3.1, the start action is also a *provider* and the above condition does not have to be extended. Hence, the algorithm *extract* stays the same as before.

12.7.2 Heuristic Candidate Selection

This module requires two major changes. First, candidates that would violate a protected literal in L^{prot} cannot be chosen. Hence, we remove all action points from the candidate set that have a negation of any protected literal as its effect. However, we do not remove the end action point of the protecting action, see Line 1 in Algorithm 9. Note that these action points removed from $\mathcal{A}^{can}_{\vdash\dashv}$ will, in some future state, be again extracted from the input plans and will be potentially merged into the final plan.

Second, we need to differentiate between the start and end action points when choosing the best candidate to be merged in the current iteration due to the dependency issue. For example, if there exists $a_{\dashv} = (a, t_{\dashv})$ in the set of candidates such that $t_s = t_{\dashv}$, then $a_{\dashv}$ needs to be merged into the current state, in order to guarantee that its time constraints are respected. If it is in the applicable set, it is chosen as the candidate, see Line 9. However, if it is in the inapplicable set, backtracking must be triggered because the action point needs to merged at that moment but it requires some additional actions in order to satisfy its preconditions. Therefore, this method returns an empty set in order to trigger backtracking (Line 13).

Algorithm 9: heuristically-choose

```
Input: $\mathcal{A}_{\vdash}^{can}, L_s, \pi_s, L^{prot}, t_s$
Output: $a_{\vdash}^{can}$, $\mathcal{A}_{\vdash}^{can}$
1: [New] $\forall a_{\vdash} \in \mathcal{A}_{\vdash}^{can} \mid \exists l \in \mathit{eff}(a_{\vdash}) : (\neg l, a'_{\vdash}) \in L^{prot} \wedge a_{\vdash} \neq a'_{\dashv}$,
   $\mathcal{A}_{\vdash}^{can} = \mathcal{A}_{\vdash}^{can} \setminus \{a_{\vdash}\}$
2: $\mathcal{A}_{\vdash}^{app} = \{a_{\vdash} \in \mathcal{A}_{\vdash}^{can} \mid pre(a_{\vdash}) \subseteq L_s\}$
3: $\mathcal{A}_{\vdash}^{inapp} = \mathcal{A}_{\vdash}^{can} \setminus \mathcal{A}_{\vdash}^{app}$
4: $\Pi^{join} = \{X\,(D_i, A_i, (L_s, pre(a_{\vdash}))) \oplus a_{\vdash} \mid a_{\vdash} \in \mathcal{A}_{\vdash}^{inapp}\}$
5: if $\mathcal{A}_{\vdash}^{app} = \emptyset$ and $\Pi^{join} = \emptyset$ then
6:     return $\emptyset$, $\mathcal{A}_{\vdash}^{can}$
7: else
8:     if [New] $\exists\, a_{\dashv} \in \mathcal{A}_{\dashv}^{app} \mid t_s = t_{\dashv}$ then
9:         $a_{\vdash}^{can} = a_{\dashv}$
10:        return $a_{\vdash}^{can}$, $\mathcal{A}_{\vdash}^{can}$
11:    end if
12:    if [New] $\exists\, a_{\dashv} \in \mathcal{A}_{\dashv}^{inapp} \mid t_s = t_{\dashv}$ then
13:        return $\emptyset$, $\mathcal{A}_{\vdash}^{can}$
14:    end if
15:    $a_{\vdash}^{can}$, $\mathcal{A}_{\vdash}^{can}$ = heuristic($\mathcal{A}_{\vdash}^{app}, \Pi_{\vdash}^{join}, \pi_s$)
16:    return $a_{\vdash}^{can}$, $\mathcal{A}_{\vdash}^{can}$
17: end if
```

12.7.2.1. Heuristic and Greedy Selection Mechanism

The extension to the heuristic algorithm is described in Algorithm 10. We want to first choose as many start action points as possible, in order to merge actions in parallel to minimise the overall duration. Therefore, an action end is chosen only if there are no action starts to choose from (Line 2). We introduce new heuristic h^3 that extends the previous heuristic h^{link} by considering time. Note that π^{can} is a plan segment (Def. 12.3.9). Therefore, it is either action end point from the applicable set $\mathcal{A}_{\dashv}^{app}$ or a joining plan, where the last action point of the joining plan is the end action point from the inapplicable set. Therefore, we can obtain the last action point from the plan segment $a_{\dashv}^{last}$, which is an action end point. Because the corresponding action start point has been already merged in some previous iteration step, this action point is constrained in time. It needs to be merged at time $t_{\dashv}^{last}$. Therefore, we test if the chosen plan segment is not too long. If yes, the algorithm returns empty sets in order to trigger backtracking (Line 5). The rest of the algorithm is the same as before, with just one action point extracted from the candidate plan segment. We extract just one action point in order to increase chances of interleaving different plans.

Algorithm 10: heuristic

```
Input: $\mathcal{A}^{app}_{\vdash\dashv}, \Pi^{join}_{\vdash\dashv}, \pi_s, t_s$
Output: $a^{can}_{\vdash\dashv}$, $\mathcal{A}^{can}_{\vdash\dashv}$
1:  if [New] $\mathcal{A}^{app}_{\vdash} = \emptyset$ and $\Pi^{join}_{\vdash} = \emptyset$ then
2:      $\pi^{can} = h^3(\mathcal{A}^{app}_{\dashv}, \Pi^{join}_{\dashv})$
3:      $a^{last}_{\dashv}$=get-last-action-point($\pi^{can}$)
4:      if $|\pi^{can}| + t_s > t^{last}_{\dashv}$ then
5:          return $\emptyset, \emptyset$
6:      end if
7:  else
8:      $\pi^{can} = h^3(\mathcal{A}^{app}_{\vdash}, \Pi^{join}_{\vdash})$
9:  end if
10: $\mathcal{A}^{can}_{\vdash\dashv} = $ extract($\{\pi^{can}\}, \pi_s, \mathcal{A}^{can}_{\vdash\dashv}$)
11: $a^{can}_{\vdash\dashv} = $ rules($\mathcal{A}^{can}_{\vdash\dashv}$)
12: return $a^{can}_{\vdash\dashv}$, $\mathcal{A}^{can}_{\vdash\dashv}$
```

12.7.2.2. Heuristic Function

As the proposed heuristic h^{link} from the previous chapter does not consider time, we need to propose new heuristic that would lead to minimising the duration of the solution plan. A naïve choice to obtain the shortest makespan is the following heuristic:

h_1 : choose the shortest *plan segment*

This greedy heuristic can often lead to a situation where π_s violates many causal links in the input plans. Therefore, more actions will be *inapplicable* in the future iteration and more actions from joining plans will need to be merged, leading to prolonging the makespan. Hence, a heuristic that considers causal links can have a better performance than h_1. This is similar to heuristic h^{link} but considers time as well:

h_2 : choose the shortest *plan segment* that does not violate a causal link $a_i^m \xrightarrow{l} a_i^{unm}$ in any input plan $\pi_i^\times$ such that the *producing* action a_i^m is already merged but the the *consuming* a_i^{unm} is not.

This gives a look ahead to the heuristic based on the literals in the current state L_s that need to be preserved for a future state. However, the cost of preserving the literal from a causal link can be bigger than the cost of violating it and adding a joining plan later. This is due to the fact that this heuristic drives the merge algorithm to sequence the input plans than interleave them. The sequencing of plans occurs because links from one plan get violated as a result of an action from another plan has been merged. Moreover, this heuristic adds some computational overhead.

In order to avoid the disadvantages of the above heuristic, we propose the following heuristic:

h_3 : choose the shortest candidate plan corresponding to the *applicable* action point if it exists, or choose the shortest candidate plan corresponding to the joining plan.

This heuristic does not add any computational cost when compared to h_2, and it guides the merge to limit the occurrence of joining plans. This guidance is done in a way that the heuristic prefers longer plan segments corresponding to the applicable action point, rather than shorter joining plans. However, as the chosen applicable action can be from different plans each time, the joining plans may be still required if the causal links get violated. As a result, the interleaving of the plans is supported by this heuristic as well (similarly to h_1). We add several rules to the heuristic h_3 in order to make it deterministic while deciding a draw between two plan *segments*. The heuristic chooses the plan segment that:

1. violates fewer input causal links;

2. its plan has fewer action related flaws, i.e, that is closer to be merged completely;

3. its plan has shorter duration;

These decision rules are ordered in the way that they are applied, i.e., if a draw still occurs after the first rule is applied, the second is applied, etc.

12.7.3 Merge Algorithm

The merge algorithm needs to be extended in three different ways. First, if an action start is merged then all its invariant precondition literals must be saved as *protected*. Second, if an action end is merged, its invariant preconditions need to be removed from L^{prot}. Third, the added action point may be ordered in parallel to

Algorithm 11: merge

```
Input: $a_H^{can}, \mathcal{A}_H^{can}, S$
Output: $S$
1:  $L_s = L_s \cup \mathit{eff}(a_H^{can})$
2:  $\forall f \in \mathit{eff}^+(a_H^{can}) : a_f^+ = a_H^{can}$
3:  $\forall f \in \mathit{eff}^-(a_H^{can}) : a_f^- = a_H^{can}$
4:  $\mathcal{A}_s^{can} = \mathcal{A}_H^{can} \setminus \{a_H^{can}\}$
5:  if [New] $a_H^{can} \in \mathcal{A}_\vdash^{can}$ then
6:      $L^{prot} = L^{prot} \cup \{l\} \mid l \in pre(a_\vdash^{can})_\leftrightarrow$
7:  end if
8:  if [New] $a_H^{can} \in \mathcal{A}_\dashv^{can}$ then
9:      $L^{prot} = L^{prot} \setminus \{l\} \mid l \in pre(a_\dashv^{can})_\leftrightarrow$
10: end if
11: $\mathcal{A}_{\pi_s} = \mathcal{A}_{\pi_s} \cup \{a_H^{can}\}$
12: $\forall l \in pre^+(a_H^{can}) : \mathcal{L}_{\pi_s} \cup \langle a_l^+ \xrightarrow{l} a_H^{can} \rangle$,
    $\mathcal{O} = \mathcal{O} \cup \langle a_l^+ \prec a_H^{can} \rangle$
13: $\forall l \in pre^-(a_H^{can}) : \mathcal{L}_{\pi_s} \cup \langle a_l^- \xrightarrow{l} a_H^{can} \rangle$,
    $\mathcal{O} = \mathcal{O} \cup \langle a_l^- \prec a_H^{can} \rangle$
14: $\mathcal{O} = \mathcal{O} \cup \langle a_H^{last} < a_H^{can} \rangle$
15: [New] $\mathcal{A}^{threat} = \text{threats}(a_H^{can}, \mathcal{L}_{\pi_s})$
16: [New] $\forall a^{threat} \in \mathcal{A}^{threat} : \mathcal{O}_{\pi_s} = \mathcal{O}_{\pi_s} \cup \langle a^{threat} \prec a_H^{can} \rangle$
17: $t_s = \text{makespan}(\mathcal{O}_{\pi_s})$
18: return $S$
```

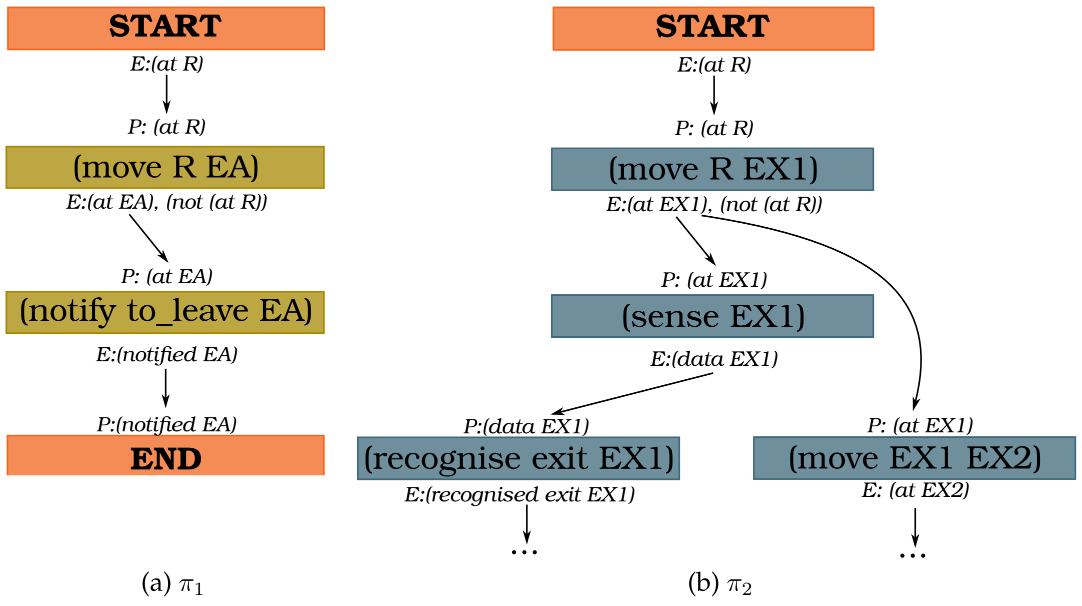


Figure 12.4: Details of partially ordered plans for task ω_1 (a) and ω_2 (b). The arrows visualise the causal links between actions. The dots represent that there are more actions in the plan.

another action and could threaten an existing causal link (line 15). In such a case, the action point is *promoted*.

12.7.4 Example

We are going to continue with the example given in Sec. 12.2. After two iterations described before, the merged plan contains action (move R EX1) and (sense EX1). In these iterations, there is no difference how POPMER would behave using instantaneous or durative actions. The makespan of this plan is $t_s = 2$ units.

The third iteration starts to be interesting while merging durative actions. The candidates are $a_{\vdash 1-1}$ = (move R EA) and $a_{\vdash 2-3}$ = (recognise exit EX1). Note that both are start action points. The heuristic module chooses $a_{\vdash 2-3}$ because it is applicable in the current state $L_s = \{$(at EX1),(data EX1)$\}$. This action point is merged, where L_s is unchanged because the start action point does not have any effects. Additionally, this action has an invariant precondition (data EX1). Therefore, this literal becomes *protected*, i.e., L^{prot} = (data EX1). The duration of the plan t_s is unchanged as the start action point has been merged. We omit the remaining details about the search state, as they are irrelevant to what we are going to illustrate.

In the fourth iteration, the candidates are $\mathcal{A}^{can}_{\vdash\!\dashv} = \{a_{\vdash 1-1}, a_{\dashv 2-3}\}$. Importantly, $a_{\vdash 1-1}$ is not *violating* the protected literal (data EX1). Furthermore, $a_{\dashv 2-3}$ does not have to be chosen due to its temporal constraints because of the current time $t_s \neq t_{\dashv 2-3}$. As a result, the heuristic can choose between both of these action points.

The heuristic prefers to choose a start action point rather than an end action point in order to perform as many actions in parallel as possible. Therefore, the start action point $a_{\vdash 1-1}$ is chosen. Because it is inapplicable in the current state (it requires (at R)), a joining plan is obtained. In this case, the joining plan includes action a_{temp-1} = (move EX1 R) and $a_{\vdash 1-1}$. However, the heuristic calls method *extract* in order to obtain only one action point from the joining plan, which is going to be merged. This is done in order to support as many merging opportunities as possible. If we would merge all the joining plan, we might miss an opportunity. Nevertheless, the exploitation of opportunities comes in the expense of a need to find more joining plans. Finally, the start action point $a_{\vdash temp-1}$ is merged. As a result, actions a_{temp-1} = (move EX1 R) and a_{2-3} = (recognise exit EX1) are ordered in parallel in the final merged plan.

12.8 Plan Merging with Durative Actions and Time Constraints

Durative tasks with time constraints add supplementary difficulty: an action point from a plan related to a task cannot be merged outside of its task time window. Therefore, the *reasoning and managing* module must reason about this extra constraint. Violating the corresponding deadline causes *backtracking*. Moreover, POPMER handle all temporal constraints explicitly. Hence, the planner X can support only PDDL 2.1. level 3. This decision is made as there are not that many planners supporting timed initial literals in PDDL 2.2 level 3, which are needed to model temporal constraints.

12.8.1 Reasoning and Managing

If the planner X does not find a plan with makespan less than the time window, then such problem is unsolvable (Line 5 of Algorithm 12). Therefore, this module returns an empty set of candidates that triggers the backtracking module. However, as there are no decisions to backtrack, POPMER terminates with no solution found. Alternatively, if the plan does fit the given time window, the algorithm can choose only such candidates from tasks whose time window is *open*, which means that the current time t_s is within release date and deadline.

Algorithm 12: extract

```
Input: $\Omega^\times$, $\pi_s$, $\mathcal{A}_{\mathsf{H}}^{can}$, $t_s$
Output: $\mathcal{A}_{\mathsf{H}}^{can}$
 1: $a_{\mathsf{H}}^{last}$ = get-last-action-point($\pi_s$)
 2: $\mathcal{A}_{\mathsf{H}}^{can} = \mathcal{A}_{\mathsf{H}}^{can} \setminus \{a_{\mathsf{H}}^{last}\}$
 3: for $\omega_i^\times = \langle D_i, A_i, (I_i, G_i), \pi_i^\times, p_i, r_i, d_i \rangle \in \Omega^\times$ do
 4:    if [New] $|\pi_i^\times| > d_i - r_i$ then
 5:       $\emptyset$
 6:    end if
 7:    if [New]$r_i \leq t_s \leq d_i$ then
 8:       $\mathcal{A}_{\mathsf{H}}^{unm}, \mathcal{A}_{\mathsf{H}}^{m}$ =merged-action-points($\pi_i^\times$, $\pi_s$)
 9:       $\mathcal{A}_{\mathsf{H}}^{can} = \mathcal{A}_{\mathsf{H}}^{can} \cup \{a_{\mathsf{H}} \in \mathcal{A}_{\mathsf{H}}^{unm} \mid a_{\mathsf{H}} \notin \mathcal{A}_{\mathsf{H}}^{can} \wedge (\forall a'_{\mathsf{H}} \in \mathcal{A}_{a_{\mathsf{H}}}^{dep} : a'_{\mathsf{H}} \in \mathcal{A}_{\mathsf{H}}^{m})\}$
10:    end if
11: end for
12: return $\mathcal{A}_{\mathsf{H}}^{can}$
```

12.8.2 Heuristic Candidate Selection

The novelty in the heuristics algorithm, see Algorithm 13, is that the chosen plan segment π^{can} must fit the time window of the corresponding task. If it is too long, an empty set is returned, in order to trigger backtracking.

Algorithm 13: heuristics

```
Input: $\mathcal{A}_{\mathsf{H}}^{app}, \Pi_{\mathsf{H}}^{join}, \pi_s, t_s$
Output: $a_{\mathsf{H}}^{can}$, $\mathcal{A}_{\mathsf{H}}^{can}$
 1: if $\mathcal{A}_{\vdash}^{app} = \emptyset$ and $\Pi_{\vdash}^{join} = \emptyset$ then
 2:    $\pi^{can} = h^3(\mathcal{A}_{\dashv}^{app}, \Pi_{\dashv}^{join})$
 3:    $a_{\dashv}^{last}$=get-last-action-point($\pi^{can}$)
 4:    if $|\pi^{can}| + t_s > t_{\dashv}^{last}$ then
 5:       return $\emptyset, \emptyset$
 6:    end if
 7: else
 8:    $\pi^{can} = h^3(\mathcal{A}_{\vdash}^{app}, \Pi_{\vdash}^{join})$
 9: end if
10: $d$ = get-deadline-of-corresponding-task($\pi^{can}$)
11: if $|\pi^{can}| + t_s > d$ then
12:    return $\emptyset, \emptyset$
13: end if
14: $\mathcal{A}_{\mathsf{H}}^{can} =$ extract($\{\pi^{can}\}, \pi_s, \mathcal{A}_{\mathsf{H}}^{can}$)
15: $a_{\mathsf{H}}^{can} =$ rules($\mathcal{A}_{\mathsf{H}}^{can}$)
16: return $a_{\mathsf{H}}^{can}$, $\mathcal{A}_{\mathsf{H}}^{can}$
```

12.9 POPMER$_X$ Summary

Finally, as all components have been introduced, we can summarise the overall proposed algorithm POPMER$_X$, see Algorithm 14. As previously stated, the preprocessing is not required for the algorithm functionality. Hence, the first input parameter to this algorithm is the set of tasks Ω and the second is a boolean parameter on whether preprocessing should be used or not. Then, plans achieving each task are obtained separately by the wrapped planner X on Line 6. Before triggering the third layer functionality, the search state is initialised as follows.

The candidate plan π_s consists only of the START action, see Def. 10.4.2, and orderings and causal links are initialised as empty sets. Additionally, the literals L_s are initialised by those from the initial state I. Moreover, all achievers and deleters saved in $\mathcal{A}_0^+, \mathcal{A}_0^-$, respectively, are just the START action. Finally, the candidate set is empty.

The main loop of the plan merging procedure proceeds as follows. The *flaw check* on Line 8 stops the iterations when there are no goal related flaws, i.e., all goals are satisfied. Otherwise, the algorithm *extracts* potential candidates from the input plans $\Pi^\times$. If none can be extracted, the *backtracking* recovers the previous state and removes the invalid choice from the recovered candidates. If the initial state is reached by backtracking, the plans cannot be merged while maintaining their individual constraints. Therefore, POPMER$_X$ returns an empty plan as no solution can be found. In the case where some candidates have been extracted, the algorithm *heuristically chooses* the best candidate from the *applicable* or *inapplicable* action points. In the case when an inapplicable action point is chosen, the joining plan is added in order to change the state L_s in order to satisfy all preconditions of the inapplicable action point. If no candidate has been extracted, it backtracks until a candidate can be chosen. Once a candidate action point is obtained, it is *merged*, updating the search state according to its effects, and duration (in the case of end action points).

Algorithm 14: $POPMER_X(\Omega, preprocess)$

```
 1: if $preprocess$ then
 2:    $\Omega^\times = \text{preprocess}(\Omega)$
 3: else
 4:    $\Omega^\times = \Omega$
 5: end if
 6: $\Pi^\times = X(\Omega^\times)$
 7: $S = \langle I, \mathcal{A}_0^+, \mathcal{A}_0^-, \emptyset, \langle\{\textsf{START}\}, \emptyset, \emptyset\rangle\rangle$
 8: while $\exists\, \gamma_G$ do
 9:    $\mathcal{A}_{\textsf{H}}^{can}$ = extract($\Pi^\times, \pi_s, \emptyset$)
10:    while $\mathcal{A}_{\textsf{H}}^{can} = \emptyset$ do
11:       $\mathcal{A}_{\textsf{H}}^{can}, S$ = backtrack($S$)
12:       if $S = \langle \mathcal{I}, \mathcal{A}_0^+, \mathcal{A}_0^-, \emptyset, \langle\{\ \textsf{START}\}, \emptyset, \emptyset\rangle\rangle$ then
13:          return $\langle\emptyset, \emptyset, \emptyset\rangle$
14:       end if
15:    end while
16:    repeat
17:       $a_{\textsf{H}}^{can}, \mathcal{A}_{\textsf{H}}^{can}$ = heuristically-choose($\mathcal{A}_{\textsf{H}}^{can}, L_s, \pi_s$)
18:       if $a_{\textsf{H}}^{can} = \emptyset$ then
19:          $\mathcal{A}_{\textsf{H}}^{can}, S$ = backtrack($S$)
20:          if $S = \langle \mathcal{I}, \mathcal{A}_0^+, \mathcal{A}_0^-, \emptyset, \langle\{\ \textsf{START}\}, \emptyset, \emptyset\rangle\rangle$ then
21:             return $\langle\emptyset, \emptyset, \emptyset\rangle$
22:          end if
23:       end if
24:    until $a_{\textsf{H}}^{can} = \emptyset$
25:    $S$ = merge($a_{\textsf{H}}^{can}, \mathcal{A}_{\textsf{H}}^{can}, S$)
26: end while
27: return $\pi_s$
```

CHAPTER 13

EVALUATION

Proclamation: Results presented in this chapter have been significantly improved. Please refer to our journal paper.

We have developed a version of our algorithm POPMER$_{\text{VHPOP}}$, which embeds the VHPOP planner (Simmons & Younes, 2011) for plan generation for individual tasks. Nevertheless, any temporal planner implementing PDDL 2.1 can be used within POPMER. We have chosen VHPOP as it is a *partially-ordered temporal planner* and it handles *required concurrency*. Moreover, it is *not* an anytime planner, thus it terminates after a solution is found. In the case of an anytime planner, POPMER would need to monitor the run of the embedded planner and to stop the planner after a solution was found.

Importantly, the performance of VHPOP strongly depends on the configuration of its input parameters. Therefore, we differentiate between two versions: the original VHPOP, denoted as O-VHPOP, and IPC3-VHPOP, which entered the third International Planning Competition (IPC). By the original version, we mean that VHPOP runs without any parameter. In contrast, IPC3-VHPOP uses a combination of several flaw and plan selection heuristics and the exact configuration and discussion can be found on the VHPOP GitHub page[1]. Importantly, IPC3-VHPOP is significantly faster than the original version, but it is unable to handle

[1]https://github.com/hlsyounes/vhpop

the required concurrency. If not stated otherwise, while discussing POPMER implementation we allude to the one that integrates IPC3-VHPOP.

We evaluate POPMER handling durative actions and also the version handling time constraints. We do not evaluate the instantaneous version, as our interest is in temporal domains. We evaluate both versions of POPMER separately because many existing temporal planners do not support timed initial literals used to represent time windows. However, the methodology of our evaluation is the same for both.

13.1 Methodology

Planners are often evaluated on tens of planning problems from different domains in order to measure their properties, such as average plan quality, coverage and scalability. The International Planning Competition (IPC) has introduced a relative score, $RIPCscore$, in order to compare planners. For each solved problem, either the optimal solution with makespan M^* is known or the best makespan, found by one of the compared planners, is considered. $RIPCscore$ is defined as follows:

$$RIPCscore_j^p = \frac{M^*}{M_p}, \tag{13.1}$$

where j refers to the solved planning problem and M_p is the makespan of the plan found by the compared planner p. Hence, the best planner obtains a score of one and poorer planners have a score smaller than one. A zero score corresponds to the case when a planner did not find a solution.

We have decided to slightly modify this criterion due to two reasons. First, we do not have the optimal solution to the problems. Second, we are interested in comparing POPMER$_X$ with other planners. Therefore, we use the quality found by POPMER$_X$ as M^*. Hence, if the $RIPCscore^p_j$ is smaller than one, the compared planner p found a poorer solution. If it is more than one, it has found a better solution in comparison to POPMER$_X$. For anytime planners, which can provide more solutions, we consider the latest one, i.e., the one with the shortest makespan. For stochastic planners, we run ten instances with different seeds and we consider the best makespans from these instances.

Furthermore, we report on the distribution of $RIPCscores$ on a domain by presenting three quartiles Q_1, Q_2, Q_3. Additionally, the IPC proposed an absolute score of a planner in order to express overall performance of the planner. It is obtained as a sum of the relative IPC scores over all of tested problems:

$$AIPCscore^p = \sum_{j \in \{1 \ldots m\}} RIPCscore^p_j, \tag{13.2}$$

where m is the number of planning problems presented to the planner. Hence, scores bigger than value m are better than POPMER$_X$, while scores smaller than m are not as good.

Another important aspect of the planner performance is coverage. It represents whether a solution has been found and if it is valid. We evaluate the found plans by utilising VAL, the validator of PDDL plans[1], in order to ensure that the plan is valid for the domain and problem. A planner's coverage on a domain is expressed by the number of solved problems from that domain within the given time and memory limits.

$$\begin{aligned} C^p &= 0 \\ \forall j \in \{1 \ldots m\} : C^p &\mathrel{+}= \begin{cases} 1 & \text{if } RIPCscore^p_j > 0 \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{13.3}$$

Coverage is tightly related to scalability, which expresses how successful a

[1]http://www.inf.kcl.ac.uk/research/groups/PLANNING/

planner is on covering a broad range of representative problems. Recall that one of the motivations of this thesis is to address scalability of temporal planning techniques. Therefore, we evaluate how scalability of a planner is dependent on runtime or consumed memory. Hence, we measure runtime and a peak of memory needed to find a solution by a planner. The time measurement is rounded to tenths of seconds and the memory is rounded to units of kB.

There may be different ways to make a problem harder for each planner. The IPC defines a harder problem as the one with more goal literals. However, a planning problem with more goal literals does not necessarily have to be harder for a planner, compared to another with fewer goals. Therefore, the scalability of a planner does not have to be an increasing function with respect to number of goals.

13.2 Planning with Durative Actions

Because IPC3-VHPOP is unable to handle required concurrency, we use two versions of POPMER in this comparison: POPMER$_{\text{O-VHPOP}}$ or POPMER$_{\text{IPC3-VHPOP}}$. We compared those to various state-of-the-art planners, for which we were able to obtain their source-code or binaries. Because the IPC 2014 provided precompiled binaries for planners, most of the compared planners are from IPC 2014[1]. We have added VHPOP and OPTIC in order to cover all different approaches for temporal planning, that were discussed previously in the related work (Sec. 9.1). All planners, which are going to be used in this comparison, were introduced in the related work chapter as well. Here, we briefly remind their most important properties. They can be split into two groups according to their support of required concurrency:

- Not supporting required concurrency:

[1]https://helios.hud.ac.uk/scommv/IPC-14/

    - DAE$_X$ utilises a genetic algorithm in order to split a planning problem to smaller instances that are solved by a wrapped planner X. We report on DAE$_{\text{YAHSP2}}$.
    - YAHSP3 is a first fit planner. It simplifies durative actions to instantaneous ones.
    - TFD is a decision epoch planner.

- Supporting required concurrency:
    - OPTIC is a temporally-lifted planner, which handles durative actions in STNs. We use OPTIC with CPLEX solver.
    - ITSAT solves planning as a satisfiability problem.
    - VHPOP is a backward-chaining POP planner, it utilises STNs as well.

All these planners solve the *conjoined* problem.

13.2.1 Domains and Problems

The planners are compared on different domains taken from IPC 2014. However, we generate our own planning problems in these domains, as our algorithm is based on the assumptions that tasks (i.e., goals in problems) are separable. Moreover, we assume that only a single agent is able to perform the actions. Furthermore, we need to generate equivalent sets of problems to feed into our algorithm and into other planners performing the conjoining or sequencing strategies.

Problem files are generated for the conjoining approach first, as their problems are standard within the community and there exist problem generators. Each problem can have multiple goals, say n goals. Then, one conjoining problem file is split into n merging problem files, where each contains only one particular goal.

Furthermore, the initial state of merging and conjoining problems is the same. Finally, problem files for the sequencing approach are generated from the merging files while performing the sequencing approach.

The sequencing approach first chooses one problem from the set of merging problems (by a FIFO strategy) and a planner obtains a plan for it. Then, the final goal state reached by a planner is written as the initial state of the new problem. This process continues until n plans are obtained. Finally, the plans are concatenated into a sequence.

Because each of the approaches provide a single plan fulfilling all goals specified in the conjoining problem, we are able to compare their performance. All the problems we use in our evaluation are available online [1].

13.2.1.1. Overview of Domains

In this evaluation, we use the following domains:

- The **Driverlog** domain requires that packages are moved by trucks between different locations. In the original domain, a truck needs a driver in order to operate. However, in our case, we assume that a single truck has already been boarded by a driver.
- In the **Temporal Machine Shop (TMS)** domain, different pieces of ceramic are made using a kiln, which represents the single agent.

One may wonder why do we evaluate only on these two domains. Because POPMER$_X$ is not intended to be a general planner, we focus on evaluating on the domain which is the closest to our main focus on mobile service robots – the Driverlog. We could have written our own domain and problems, but in order to avoid modelling mistakes and to guarantee soundness of the evaluation, we have chosen this well-known domain because it has spatial constraints. Hence,

[1]https://github.com/mudrole1/POPMER/tree/master/testing_files

we provide a detailed study of POPMER and other planners' behaviours on this domain with and without time windows. Finally, as this domain does not require concurrency, we have added the TMS domain, which demands *required concurrency* between actions, in order to illustrate that POPMER is capable of handling it.

13.2.2 POPMER$_{\text{VHPOP}}$ in Comparison to VHPOP

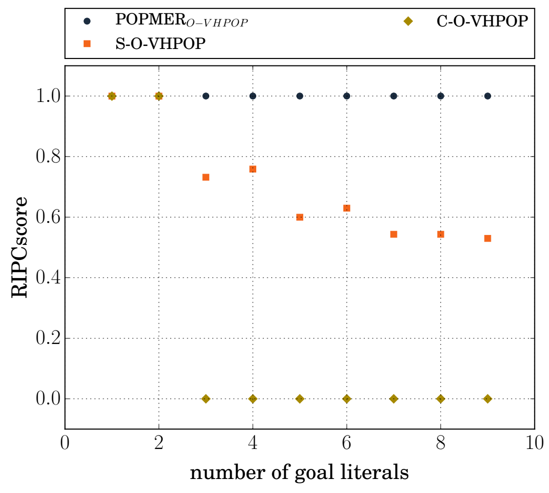
First, we analyse how performance of our proposed algorithm differs from performance of its wrapped planner, in this case VHPOP. As we employ two different versions of VHPOP, we split the comparison into two. First, we compare the original VHPOP (O-VHPOP), which solves either the conjoining (C-O-VHPOP) or the sequencing approach (S-O-VHPOP), with POPMER$_{\text{O-VHPOP}}$. Then, we compare the version of VHPOP which has participated in the third IPC (IPC3-VHPOP) in both approaches against POPMER$_{\text{IPC3-VHPOP}}$. All algorithms were run on problems for the Driverlog domain for 30 minutes. This limit is taken from the IPC setting. We limit the use of memory to 6 GB due to limitations of our hardware. We argue that this limit is realistic for an application where a planner is deployed on a robot.

13.2.2.1. Original VHPOP

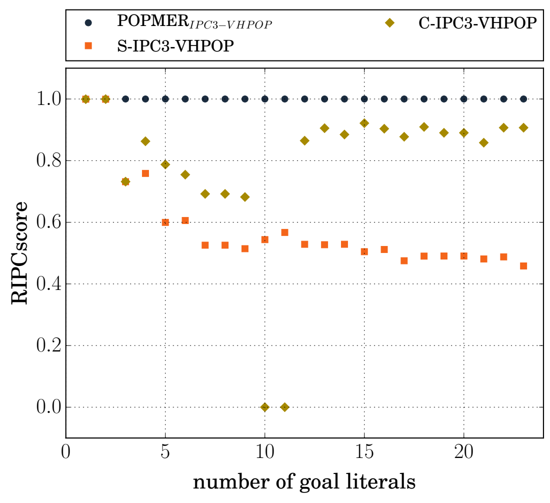
In the case of the original VHPOP, the C-O-VHPOP is able to find a solution for only two problems before it reaches the memory limit. On the successfully solved problems, it finds a plan with the same makespan as POPMER$_{\text{O-VHPOP}}$ (see Fig. 13.1a). For the unsolved problems, the $RIPCscore$ is zero. In contrast, POPMER$_{\text{O-VHPOP}}$ finds solutions for nine problems in total before it reaches the time limit. This illustrates that POPMER can improve coverage of its wrapped planner. Finally, S-O-VHPOP has significantly worse $RIPCscores$ than POPMER, on average by $28\,\%$, as expected.

Additionally, we report on the consumed memory (Fig. 13.1c). C-O-VHPOP runs out of the memory very quickly, whereas $\text{POPMER}_{\text{O-VHPOP}}$ and S-O-VHPOP are more memory efficient. Furthermore, $\text{POPMER}_{\text{O-VHPOP}}$ uses 20 times less memory than S-O-VHPOP. In regards to runtime of the planners (Fig. 13.1e), POPMER is significantly slower than S-O-VHPOP. This is mainly due to computing joining plans. Recall that POPMER can produce a temporary planning problem, which is unsolvable in the given domain, as illustrated in Sec. 12.3.2.1.. However, such a problem is not recognised by the original VHPOP, which continues searching for a solution. Therefore, we need to set a time threshold to stop the wrapped planner and to claim that the joining plan is unsolvable. Given that joining problems have typically only a few goal literals and C-O-VHPOP needs units of seconds to solve such problems, we have set this limit for 10 s. However, the runtime of $\text{POPMER}_{\text{O-VHPOP}}$ is still limited because joining plans are frequently computed.

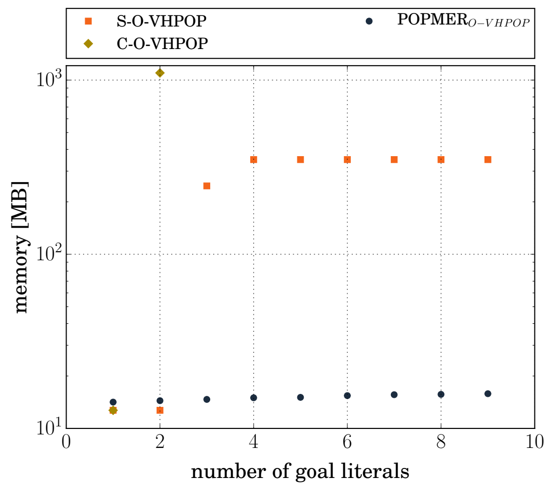
The runtime could be improved by two implementation changes to POPMER. First, recall that only one action point is extracted from a joining plan, the rest is thrown away. This leads to significant overheads. We could preserve joining plans as long as they are valid and useful. Second, we could add a detection if the generated temporary problem has an infeasible initial state, i.e., if none of domain actions can be applied. This would address the problem illustrated in Sec. 12.3.2.1. with inapplicable move action. In such a case, POPMER would not have to call the planner. Furthermore, this issue can be overcome by utilising a planner, which is able to quickly find that the temporary problem does not have a solution. This is the case for version IPC3-VHPOP.



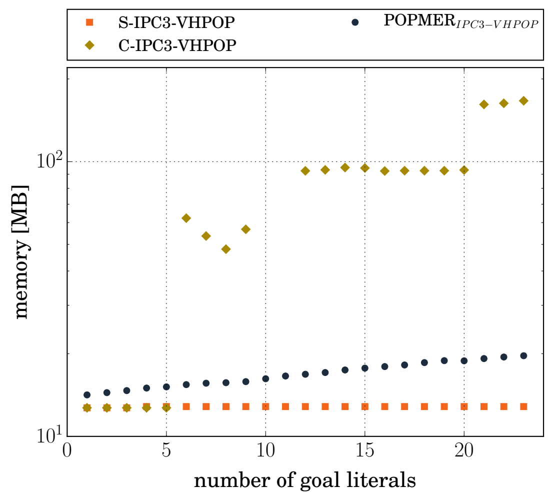
(a) O-VHPOP: *RIPCscore*



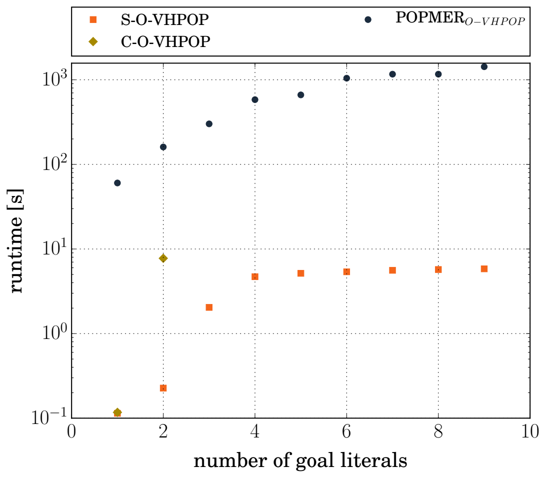
(b) IPC3-VHPOP: *RIPCscore*



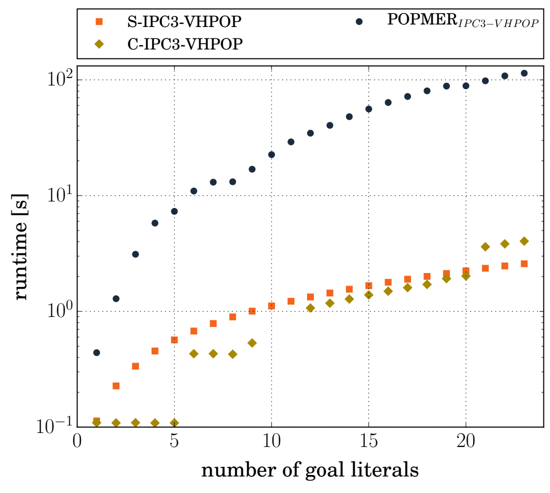
(c) O-VHPOP: memory



(d) IPC3-VHPOP: memory



(e) O-VHPOP: runtime



(f) IPC3-VHPOP: runtime

Figure 13.1: The comparison between original VHPOP (left column) and the version from the third IPC (right column).

13.2.2.2. IPC3-VHPOP

The IPC3-VHPOP employs a variety of heuristics, which significantly improves the overall performance of VHPOP. When used in the conjoining approach, not only it is able to find high quality solutions (Fig. 13.1b), but also it is very fast (Fig. 13.1f). However, it can still struggle to find a solution on some problems, as in the case of two problems with 10 and 11 goal literals. In both cases, the IPC3-VHPOP reaches the given memory limit. As a result, $\text{POPMER}_{\text{IPC3-VHPOP}}$ improves the coverage of the planner while preserving good memory scalability (Fig. 13.1d). Additionally, $\text{POPMER}_{\text{IPC3-VHPOP}}$ has found the best quality of the solution plans because:

- IPC3-VHPOP is not optimal planner. It uses several heuristics to guide the search. Especially on problems with more goal literals, the search may be guided to local optimum;
- POPMER uses IPC3-VHPOP on problems with a few literals, where IPC3-VHPOP performs well;
- POPMER utilises the preprocessing step, which extracts away demands (spatial constraints at this domain). Therefore, it can merge loading and unloading of packages at the same location for different tasks.

Furthermore, $\text{POPMER}_{\text{IPC3-VHPOP}}$ is significantly faster than $\text{POPMER}_{\text{O-VHPOP}}$ due to the fact that IPC3-VHPOP detects invalid problems in tenths of seconds. However, it could still make use of the two proposed implementation changes to get faster.

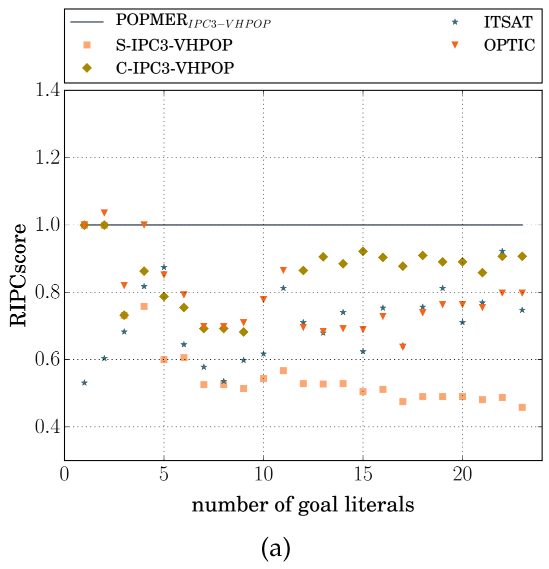
13.2.2.3. Summary

We have illustrated that POPMER_X improves the coverage, plan quality and the consumed memory of the VHPOP in two versions: original and the version from the third IPC. The runtime of POPMER could be improved by the two proposed implementation changes.

13.2.3 POPMER$_{\text{IPC3-VHPOP}}$ on the Driverlog Domain

13.2.3.1. Scores Depending on a Problem Size

For each problem from the Driverlog domain, the listed planners were run for 30 minutes and were limited to consume at maximum 6 GB of memory. The $RIPCscores$ for the planners are visualised in Fig. 13.2a and Fig. 13.2b, where we follow the separation of the planners into two groups, depending on whether they support required concurrency or not. Even though $RIPCscores$ for POPMER is not a continuous function, we visualise it as a line in order to highlight this boundary. Any score above the line is better than POPMER, whereas any under the line is worse. Moreover, we add scores for IPC3-VHPOP using the sequencing approach in order to highlight the worst estimate in both comparisons. However, some planners find poorer solutions than this approach for a number of problems.



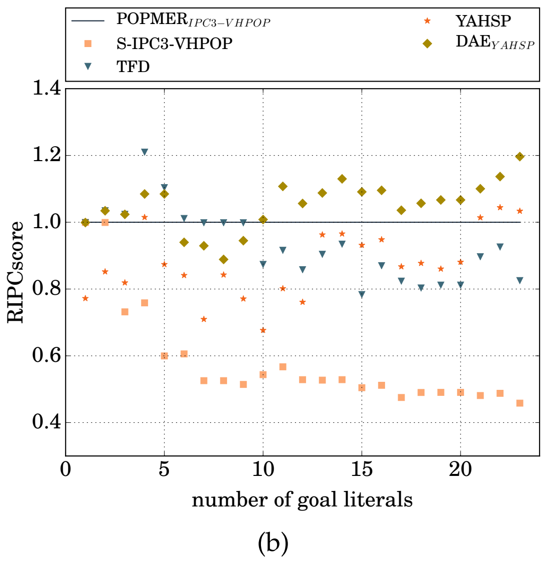


Figure 13.2: The comparison of $RIPCscores$ for planners, which (a) support the required concurrency and (b) do not, on the Driverlog domain.

	DAE	POPMER	TFD	YAHSP	VHPOP seq.
$AIPCscore$	24.17	23.00	21.42	20.12	13.34
C	23	23	23	23	23
Q_1 [%]	2	-	-16	-19	-51
Q_2 [%]	7	-	-8	-13	- 47
Q_3 [%]	9	-	0	-4	-42

	POPMER	OPTIC	VHPOP uni.	ITSAT
$AIPCscore$	23.00	18.00	17.92	16.16
C	23	23	21	23
Q_1 [%]	-	-30	-26	-38
Q_2 [%]	-	-24	-12	-29
Q_3 [%]	-	-19	-9	-24

Table 13.1: The comparison of $AIPCscore$, coverage and three quartiles of $RIPCscores$.

Table 13.1 gives an overview of $AIPCscore$, coverage C, the three quartiles of $RIPCscore$. On the one hand, according to $AIPCscore$, POPMER$_{\text{IPC3-VHPOP}}$ is the best in comparison to planners supporting the required concurrency. It significantly outperforms OPTIC and ITSAT, on average by $22\,\%$ and $30\,\%$, respectively. On the other hand, it is not the best planner in the other group, where DAE$_{\text{YAHSP}}$ surpasses it, on average by $5\,\%$.

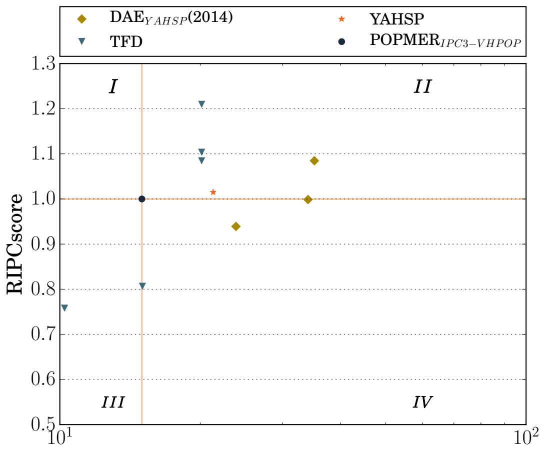
Furthermore, the score of DAE$_{\text{YAHSP}}$ is created by the best scores found by ten different instances of DAE, which were run using different seeds. Our algorithm is better than DAE$_{\text{YAHSP}}$ in four problems by an average difference per plan of 10.63 units in which makespan is recorded. As all packages must be loaded and unloaded in both plans, POPMER$_{\text{IPC3-VHPOP}}$ has only two options to find a better plan: to place loading and unloading actions concurrently or to find better path between locations. In 18 cases, POPMER$_{\text{IPC3-VHPOP}}$ found a poorer solution, when compared to the best DAE$_{\text{YAHSP}}$ by an average difference of 22.36 units. Most of these cases were problems with more goals, which is expected as the greedy heuristics are driven to local optima more often in bigger problems.

13.2.3.2. Scores Depending On Runtime and Memory

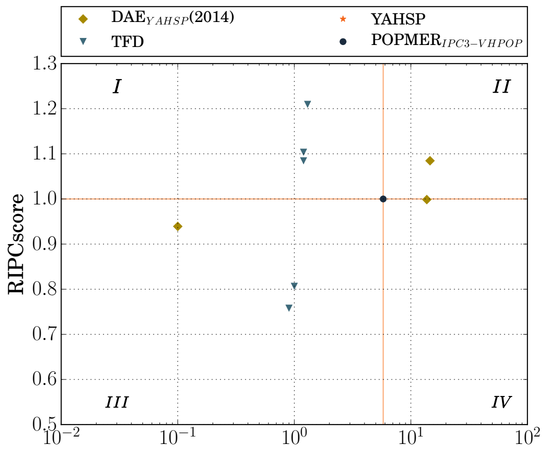
DAE$_{\text{YAHSP}}$, YAHSP and TFD find a better solution than POPMER$_{\text{IPC3-VHPOP}}$ on several problems. Therefore, we are interested in how much time and memory they use in order to provide these solutions. Additionally, as these planners are *anytime* (they provide more than one solution), we want to observe how these planners improve with increased runtime or memory. For example, a planner might provide several plans with a bad score very quickly, but it can consume a significant amount of time and memory before it reaches a better solution. Therefore, we consider all solutions found by these planners in order to observe the overall behaviour of the planners.

We have chosen three problems to analyse: with 4, 14 and 22 goal literals. These problems were chosen because all three planners score very well on them. Furthermore, we do not visualise all data related to all instances of DAE$_{\text{YAHSP}}$ because the graphs will become unclear. Therefore, we chose one particular instance of DAE with seed 2014. This configuration participated in the latest IPC. Finally, the dependencies between runtime/memory and $RIPCscore$ are visualised in Fig. 13.3. Each graph is split into four regions according to the position of the $RIPCscore$ of POPMER$_{\text{IPC3-VHPOP}}$. If a planner scores within region:

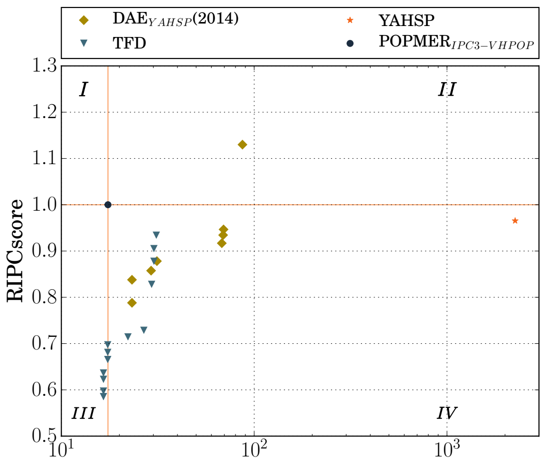
- I, then the planner has not only been faster, but has also found a better solution than POPMER$_{\text{IPC3-VHPOP}}$;
- II, then the planner has been slower, but has found a better solution than POPMER$_{\text{IPC3-VHPOP}}$;
- III, then even though the planner has been faster, it has found a poorer solution than POPMER$_{\text{IPC3-VHPOP}}$;
- IV, then the planner has been not only slower, but also found a poorer solution than POPMER$_{\text{IPC3-VHPOP}}$.



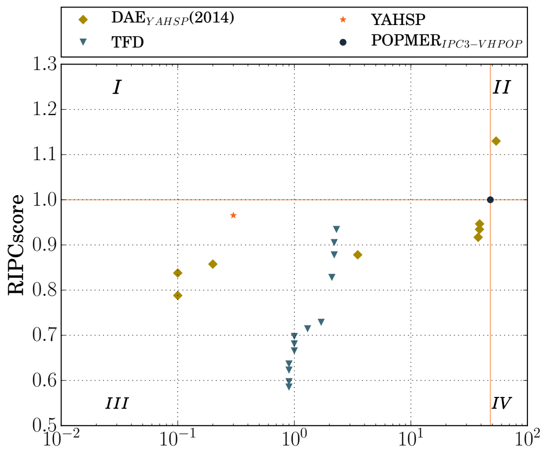
(a) a problem with 4 goal literals



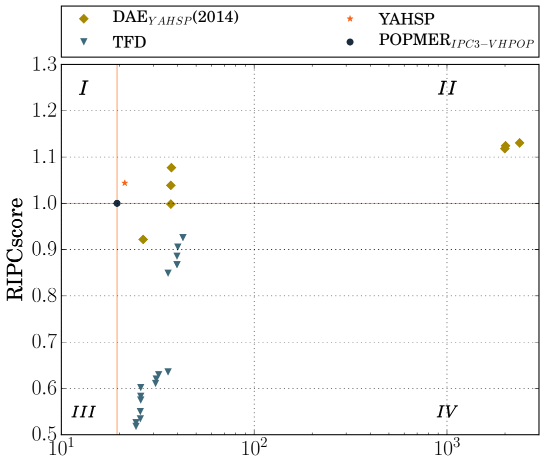
(b) a problem with 4 goal literals



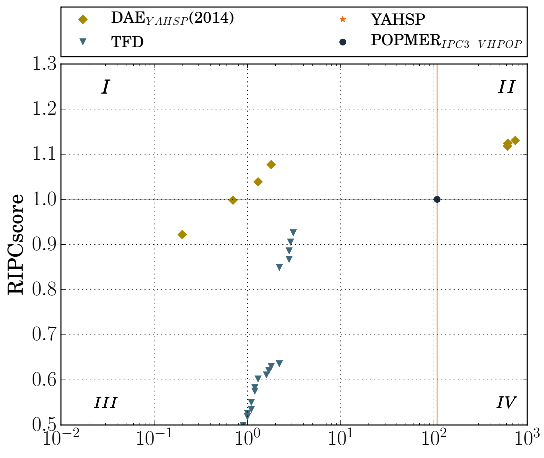
(c) a problem with 14 goal literals



(d) a problem with 14 goal literals



(e) a problem with 22 goal literals



(f) a problem with 22 goal literals

Figure 13.3: The dependencies between consumed memory/runtime and *RIPCscores* of all solutions provided by DAE$_{\text{YAHSP}}$, YAHSP and TFD.

The most important region is I as it represents where the performance of POPMER$_{\text{IPC3-VHPOP}}$ is poor. In contrast, planner entries in the region IV represent where POPMER$_{\text{IPC3-VHPOP}}$ performs best.

As we have discussed in the previous comparison, POPMER$_{\text{IPC3-VHPOP}}$ is not as fast as it could be. Hence, there is no surprise that TFD and DAE$_{\text{YAHSP}}$ score in the region I for the problem of 4 and 22 goal literals, (Fig. 13.3b and Fig. 13.3f, respectively). However, POPMER$_{\text{IPC3-VHPOP}}$ is competitive with the other planners according to the used memory. In all of the examined examples, none of the planners score in the region I. Moreover, other planners tend to score in the region IV for bigger problems.

13.2.3.3. Scalability Depending on Memory and Runtime

Finally, we report on the consumed memory and runtime of POPMER$_{\text{IPC3-VHPOP}}$ and the other planners (Fig. 13.4). Similarly, as in the analysis of $RIPCscore$, we use a continuous function for the values reached by POPMER in order to highlight this boundary (even though it is not a continuous function). When performing this evaluation, we encountered a difficulty with the OPTIC planner. Even though it is an anytime planner, it provides all its solutions into a single file. Therefore, we were unable to monitor runtime and memory used for producing a single solution. As a result, we report only on the runtime and memory needed to find the best (last) solution, which is more or less stable over all problems.

We draw similar conclusions as in the comparison of POPMER$_{\text{IPC3-VHPOP}}$ to VHPOP. That is: POPMER$_{\text{IPC3-VHPOP}}$ takes tens of seconds to provide a solution, while TFD takes units. Therefore, the runtime of POP-MER$_{\text{IPC3-VHPOP}}$ could be improved. However, a benefit of POPMER$_{\text{IPC3-VHPOP}}$ is its *deterministic* behaviour, which allows us to predict runtime on a problem. Such a feature is beneficial in order to reserve time to compute a new plan in our intended domain, where tasks

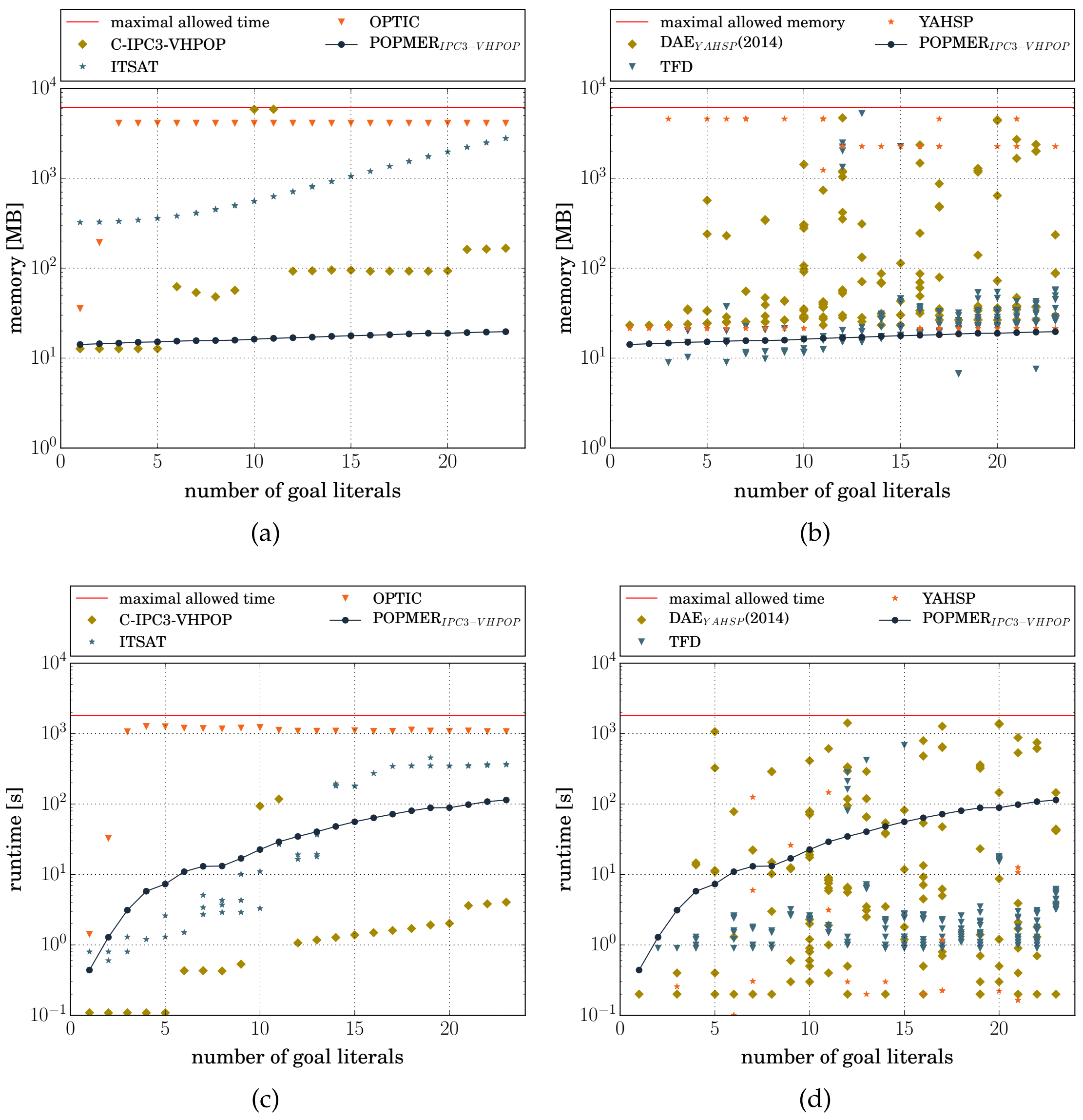


Figure 13.4: Memory and time scalability of different planners

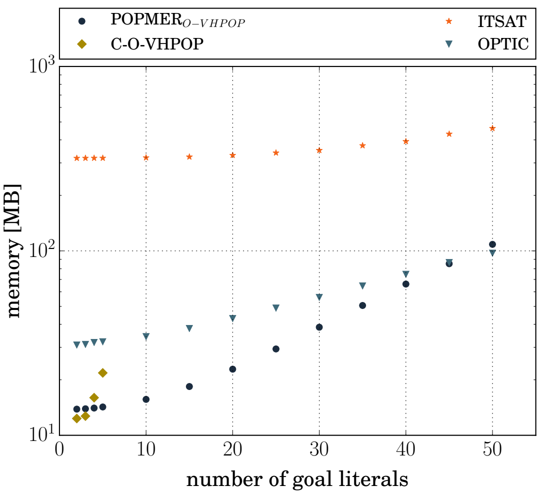
are processed in batches.

Nevertheless, POPMER$_{\text{IPC3-VHPOP}}$ demonstrates very good memory scalability, where only TFD provides some solutions with less memory used. Interestingly, YAHSP finds the first solutions using very little of memory (about 11 MB), but then it expands a lot, reaching up to 1400 MB. It is prevented from finding any other solutions because it reaches the memory limit of 6 GB. In contrast, DAE$_{\text{YAHSP}}$ reaches new solutions while expanding the memory slower than YAHSP.

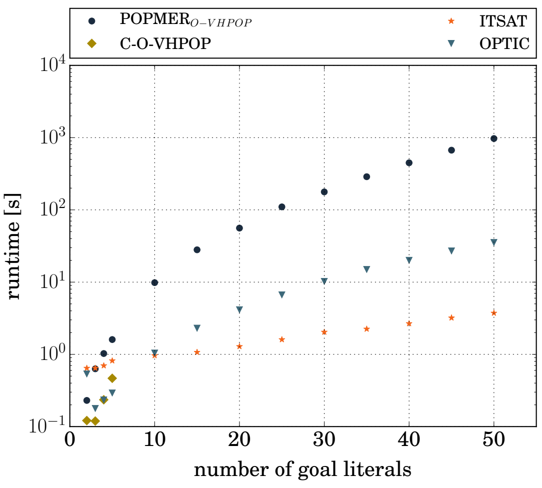
13.2.4 POPMER$_{\text{O-VHPOP}}$ on the TMS domain

TMS domain requires concurrency between actions. Therefore, we compare POPMER$_{\text{O-VHPOP}}$ only with OPTIC, ITSAT and the original VHPOP. An interesting phenomena occurs for our problems: all planners find the same makespans on the solved problems. This is because the kiln, which is used for baking ceramic, has no resource limits. Thus all the ceramic pieces can be baked in parallel. Furthermore, OPTIC and ITSAT manage to find the optimal solution immediately, without iterations. As a result, we do not report on the scores as we did above. In regards to the coverage, O-VHPOP was able to solve only 4 problems before it reached the memory limit 6 GB. In contrast, all other planners have found solutions to all 13 problems.

This experiment supports the previously made claims. First, POPMER$_{\text{O-VHPOP}}$ improves not only coverage but also scalability of the wrapped planner O-VHPOP. Second, POPMER$_{\text{O-VHPOP}}$ is significantly slower as we employ O-VHPOP, which fails to recognise invalid planning problems quickly. Third, POPMER$_{\text{O-VHPOP}}$ used the least amount of memory on the tested problem (Fig. 13.5a). However, its used memory increases with a bigger gradient than ITSAT or OPTIC on these prob-



(a)



(b)

Figure 13.5: Memory and time scalability of different planners

lems. This could be improved by limiting the use of joining plans. Moreover, these problems are trivial for OPTIC and ITSAT. Nevertheless, we can observe how POPMER$_{\text{O-VHPOP}}$ significantly improves on the used memory in comparison to O-VHPOP.

13.2.5 Summary

We have illustrated how POPMER improves on the coverage, scores and consumed memory of both O-VHPOP and IPC3-VHPOP. Its runtime could be improved by two proposed implementations changes limiting not only the computation of joining plans, but also avoiding problems with unsolvable initial states. In comparison to the other planners from IPC 2014 and OPTIC on the Driverlog domain, POPMER$_{\text{IPC3-VHPOP}}$ scores second in regards to $AIPCscore$ after DAE$_{\text{YAHSP}}$ with average difference $5\,\%$. This is caused mainly by a poorer performance of POPMER on problems with more goals, as the greedy heuristics are driven to local optima more often in bigger problems. However, DAE$_{\text{YAHSP}}$ is not able to handle problems with required concurrency whereas POPMER$_{\text{O-VHPOP}}$ is.

13.3 Planning with Durative Actions and Time Constraints

In this evaluation, we focus only on the Driverlog domain due to two reasons. First, it is the most closely related domain to our mobile robot scenario. Second, makespans strongly vary based on the travelled distance, in contrast to the TMS domain. Therefore, small time windows of tasks may result in some problems being unsolvable. This adds difficulty to the planners and it is interesting to compare how planners handle unsolvable problems.

We are able to compare POPMER$_X$ only with OPTIC as it is the only planner (from those previously considered) that supports not only PDDL 2.2 level 3 but also PDDL 3.1.

13.3.1 Domains and Problems

We extend the problems generated in the previous comparison in order to handle deadlines and release dates. PDDL 3.1 supports modelling of deadlines by the goal constraint parameter *(within d)*, which means that a goal must be reached before the deadline d. However, this parameter does not support release dates other than $r = 0$. Therefore, we use *timed initial literals* in PDDL 2.2 level 3 in order to model other release dates bigger than zero. Moreover, POPMER$_X$ utilises PDDL 2.1 and handles the time windows explicitly within the algorithm. The difference between these three representations raises the issue that each of them require a slightly different modelling of a domain and problems.

13.3.1.1. Modelling of Problems

There are a few difficulties in how to model the release dates and deadlines of tasks in three versions of PDDL. We illustrate how we model them using task: ω_1 : "Package1 needs to be delivered at location4 between two time stamps: 80 and 180 time units".

In order to model this problem in PDDL 2.1, we build on the official domain from the IPC 2014. The task goal is expressed as literal (at package1 loc4). This literal is achieved by action (unload package1 loc4). The temporal constraints are specified separately within the task specification given to POPMER$_X$.

In order to model problems in PDDL 2.2, we build on the domains from the IPC 2006. These domains use *timed initial literals* to model deadlines. For example, the following literals are added to the initial state:

(deliverable package1 loc4)

(at 180 (not (deliverable package1 loc4))).

Note that this does not consider release dates, which are assumed to be equal to zero time units.

The new predicate deliverable is added because literal (at 180 (not (at package1 loc4))) implies that package1 disappears from the loc4 at that time. In addition to these new literals, the goal is changed to (delivered package1 loc4). Furthermore, the domain is extended by a new action (deliver ?package ?loc), which has (deliverable ?package ?loc) as its precondition and (delivered ?package ?loc) as its effect.

Finally, modelling of the task's problem in PDDL 3.1 follows the same logic. However, the timed initial literals in the initial state are replaced by the within constraint of the goal.

Release dates bigger than zero cannot be modelled in PDDL 3.1. Hence, we need to use PDDL 2.2 for modelling. The following timed initial literals are added to the initial state.

(not (deliverable package1 loc4))
(at 80 (deliverable package1 loc4))
(at 180 (not (deliverable package1 loc4)))

However, these literals only restrict the occurrence of the last action (deliver package1 loc4), not of all the actions within the plan. Therefore, it can happen that package1 is unloaded at loc4 at time 14, followed by a gap until time 80, when action deliver can take place. However, this is not correct outcome. The release dates represent an external reason as to why the task cannot happen sooner. Hence, in this situation, the package cannot be at loc4 before time 80. As a result, this plan will be invalid.

To overcome this issue, we extend *all* actions by a new term ?task in order to obey the timed initial literals. Moreover, we utilise literal (valid task1) rather than literal (deliverable package1 loc4) to make the problem more readable. The modelled domains and problems are online[1].

13.3.1.2. Influence of Modelling

The modelling differences lead to a possibly poorer scalability of planners on problems modelled in PDDL 2.2, in comparison to problems in PDDL 2.1. This is due to a number of ground actions in a planning problem. For an action in the domain using PDDL 2.1, there exists k ground actions. In contrast, for the same action in the domain using PDDL 2.2, there exists $k \cdot n$ ground actions, where n is the number of tasks (the goal literals in a problem). This is because all actions has an additional term ?task.

Furthermore, the $RIPCscore$ is affected as well. Plans found by planners utilising PDDL 2.2 or 3.1 contain one more action deliver for each goal literal, in comparison to plans modelled by PDDL 2.1. As the deliver action has duration 1 time unit, the makespan found by POPMER$_X$ will be always better by n units. Therefore, we modify the computation of the $RIPCscore$ as follows:

$$RIPCscore_j^p = \frac{M*}{M_p - n}. \tag{13.4}$$

13.3.2 Time Windows with Large Overlaps

In this section, we compare POPMER$_{\text{IPC3-VHPOP}}$ with OPTIC processing problems modelled in PDDL 3.1. Hence, all the release dates of tasks are equal to zero. Again, we restrict the runtime to 30 minutes and 6 GB of used memory. We generate a set with:

- large time windows, in order to guarantee that all tasks will succeed (LTW);

[1]https://github.com/mudrole1/POPMER/tree/master/testing_files

- smaller random size time windows, where some solutions might be impossible (STW);
- impossible time windows, where no solution can be found (ITW).

All tasks time windows strongly overlap. Therefore, POPMER_X needs to consider many interactions between overlapping plans and compute many joining plans. This can affect its runtime as we have seen in the previous evaluation.

13.3.2.1. Scores and Coverage

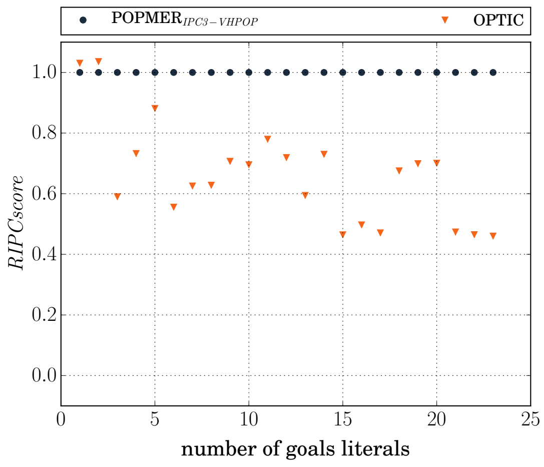
Note that problems from ITW are unsolvable. Therefore, the scores cannot be computed for these problems. Therefore, we compare in how many instances the planners terminated either:

- because they returned that no solution can be found. In such a case, we count this as a valid solution; or
- because they were terminated due to consuming too much memory or running for too long. In this situation, we count this as an invalid solution because no solution has been found.

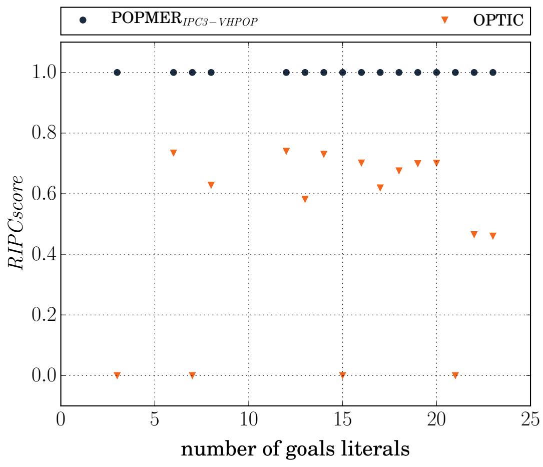
Therefore, the coverage of $\text{POPMER}_{\text{IPC3-VHPOP}}$ on the ITW instances is 23 (out of 23) because it has terminated itself stating that there is no solution. In contrast, OPTIC was able to conclude that there is no solution only in 12 instances.

On the LTW set, both planners have the same coverage (23 out of 23). On the STW set, $\text{POPMER}_{\text{IPC3-VHPOP}}$ has solved four more problems than OPTIC, i.e., POPMER's coverage is 16, whereas OPTIC's only 12. However, we are unable to state whether the remaining 7 unsolved problems are solvable or not because the time windows were generated randomly.

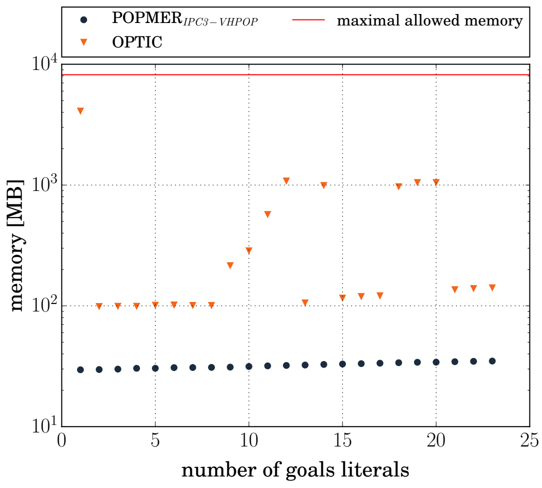
Finally, Fig. 13.6a and Fig. 13.6b compares $RIPCscores$ of $\text{POPMER}_{\text{IPC3-VHPOP}}$ and OPTIC on instances from the LTW and STW sets, respectively. The instances



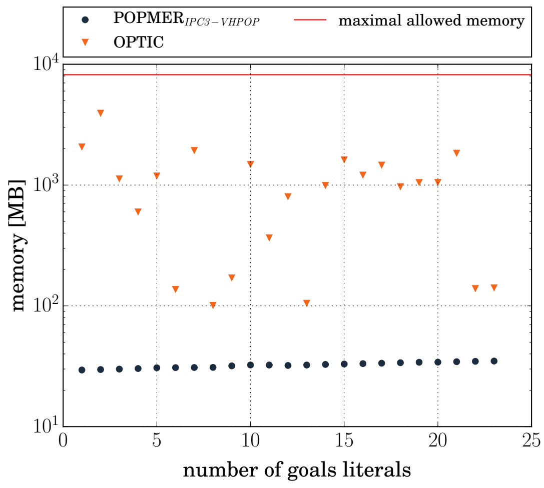
(a) Large time windows - *RIPCscore*



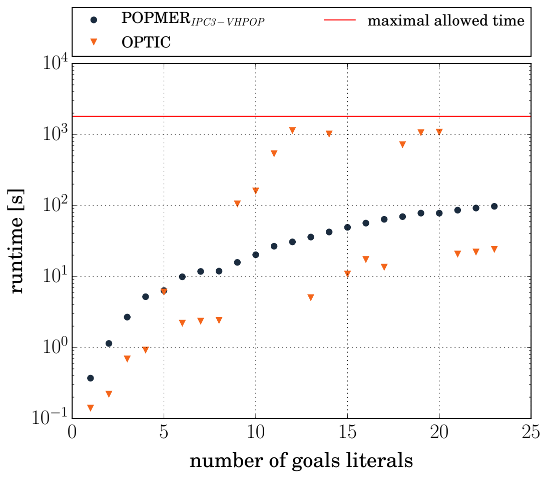
(b) Smaller time windows - *RIPCscore*



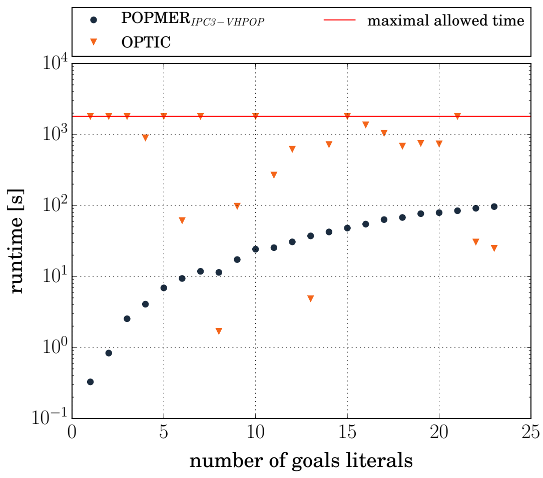
(c) Large time windows - memory



(d) Smaller time windows - memory

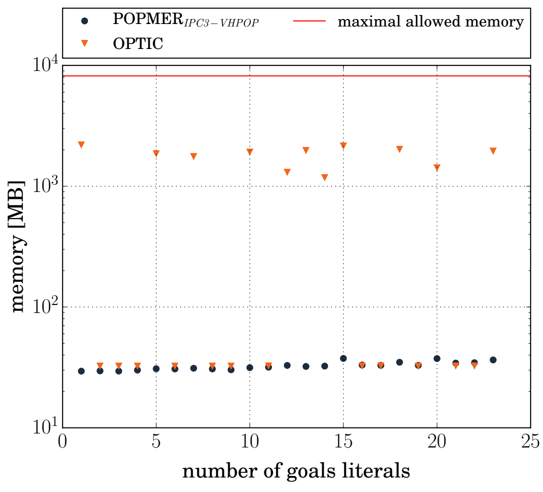


(e) Large time windows - runtime

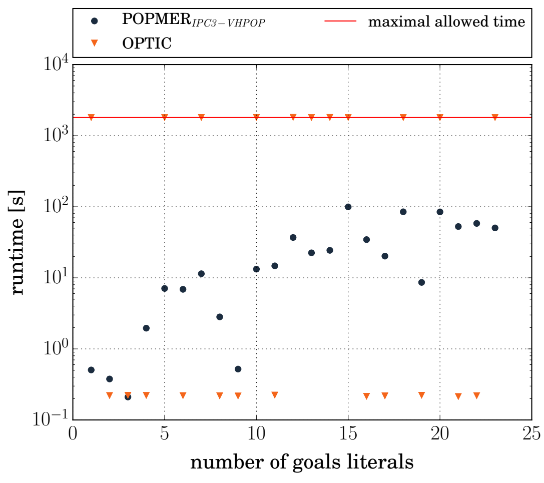


(f) Smaller time windows - runtime

Figure 13.6: A comparison of POPMER$_{\text{IPC3-VHPOP}}$ and OPTIC on the problems with with large and smaller time windows.



(a) Impossible time windows - memory



(b) Impossible time windows - runtime

Figure 13.7: The comparison of POPMER$_{\text{IPC3-VHPOP}}$ and OPTIC on the problems with with impossible time windows.

from LTW are the easiest, because planners do not backtrack too often. POPMER demonstrates very similar behaviour as on the problems with no time windows, whereas OPTIC encounters problems when it needs to backtrack. These cases are especially visible on the graphs for memory usage (Fig. 13.6c) and runtime (Fig. 13.6e), where OPTIC had difficulties in solving 8 problems. Moreover, OPTIC scores are noticeably poorer on problems encoded in PDDL 3.1 than on the problems modelled in PDDL 2.1 (Fig. 13.2a). The average difference of the $RIPCscore$ is $34\,\%$ with PDDL 3.1, while it was only $22\,\%$ with PDDL 2.1. Furthermore, on problems from STW, the differences are even bigger ($38\,\%$ on average), which is caused by the lower coverage of OPTIC, compared to POPMER (see Table 13.2 for an overview).

13.3.2.2. Scalability Depending on Runtime and Memory

Similarly as problems without time constraints, we report on the runtime of OPTIC when it found the last valid solution. The memory represents the peak of virtual memory used until OPTIC was terminated. Therefore, we are unable

	LTW	STW
$AIPCscore$	15.21 out of 23	7.73 out of 16
Q_1 [%]	-47	-65
Q_2 [%]	-32	-38
Q_3 [%]	-28	-30

Table 13.2: The comparison of $AIPCscore$ and the three quartiles of $RIPCscore$ achieved by OPTIC on problems with large (LTW) and smaller (STW) time windows.

to conclude whether either OPTIC or POPMER$_{\text{IPC3-VHPOP}}$ has better scalability on the given problems from LTW or STW. For such a conclusion, we would need to collect the runtime and memory used for all solutions found by OPTIC. However, we can still compare their overall behaviour on LTW vs STW problems.

While memory scalability of POPMER$_{\text{IPC3-VHPOP}}$ on STW problems is almost identical to the one on easier LTW problems (Fig. 13.6d and Fig. 13.6c), we see a significant increase of memory being used by OPTIC on STWs due to backtracking. A similar observation can also be done on the runtime (Fig. 13.6e and Fig. 13.6f). While POPMER$_{\text{IPC3-VHPOP}}$ demonstrates very similar behaviour on the LTWs and STWs, once again the OPTIC runtime is significantly increased on STWs, in contrast to LTWs. This illustrates how the proposed heuristic of POPMER guides its search to find good solutions quickly. Given the similarities of POPMER performance between LTW and STW problems, we can conclude that STW problems did not require much backtracking for POPMER. For example, the small difference can be observed on the problem with two goal literals, where POPMER was faster on the STW.

The extreme cases are problems from ITW where no solution is possible. On the one hand, OPTIC either recognised that no solution exists very quickly with low memory used or it needed to be terminated after 30 minutes (Fig. 13.7b). On the other hand, POPMER$_{\text{IPC3-VHPOP}}$ followed roughly the same curve as on LTW or STW. The biggest outliers are caused by the fact that a time window for a problem

	RR	SS
$AIPCscore$	12 out of 23	11.61 out of 23
Q_1 [%]	-100.0	-100.0
Q_2 [%]	-34.0	-50.0
Q_3 [%]	-2.0	0

Table 13.3: The comparison of $AIPCscore$ and the quartiles of relative scores for problems solved by OPTIC on randomly generated (RR) and sequentially generated (SR) release dates.

is too short to fit the task plan (for example problem 9).

13.3.3 Time Windows with Small Overlaps

In this section, we study the influence of release dates on the planners. Therefore, we minimise the influence of deadlines in a way that we generate large time windows in order to guarantee that a solution exists. The release dates of time windows are generated either:

- **sequentially**. This means that the release date of task ω_i is set to be larger than the deadline of task ω_{i-1}. Hence, the time windows between tasks do not overlap (SR); or
- **randomly** with some small overlaps between tasks time windows (RR).

As we model release dates that are not zero, we need to compare our algorithm with OPTIC processing problems modelled in PDDL 2.2.

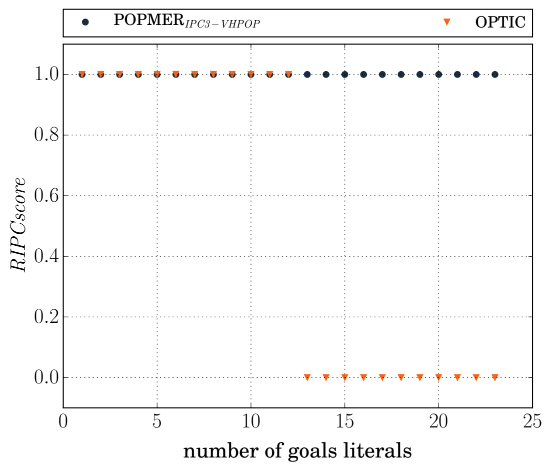
13.3.3.1. Scores and Coverage

OPTIC and POPMER find the same solutions for problems on the SR set because the tasks time windows do not overlap. Therefore, plans can start at the corresponding release dates. While POPMER$_{\text{IPC3-VHPOP}}$ found solutions for all 23 problems from both sets (SR and RR), OPTIC found only 12 and 14 solutions, respectively. On the RR set, no conclusion can be made about OPTIC scores as they are very spread out (Table 13.3).

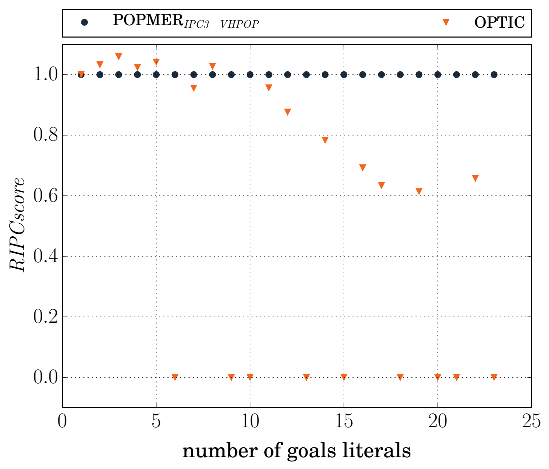
13.3.3.2. Scalability Depending on Runtime and Memory

We have argued that modelling of the problems in PDDL 2.2 may affect OPTIC's scalability. We can observe that OPTIC's runtime is strongly affected by this phenomena, see Fig. 13.8e and Fig. 13.8f. In both cases, the runtime of OPTIC is *above* POPMER's, in contrast to their runtime on the problem with large time windows modelled in PDDL 3.1 (Fig. 13.6e). Additionally, OPTIC was terminated in many cases due to its long runtime. In contrast, $\text{POPMER}_{\text{IPC3-VHPOP}}$ is ten times faster on these domains than on the LTW, STW sets or even on problems without deadlines. This is because tasks do not overlap. Hence, $\text{POPMER}_{\text{IPC3-VHPOP}}$ is able to sequence the plans. It does not have to compute any joining plan. On the RR set, we can see that $\text{POPMER}_{\text{IPC3-VHPOP}}$ is slower than on the SR set, which is caused by some task overlaps. Therefore, it needs to compute a few joining plans.

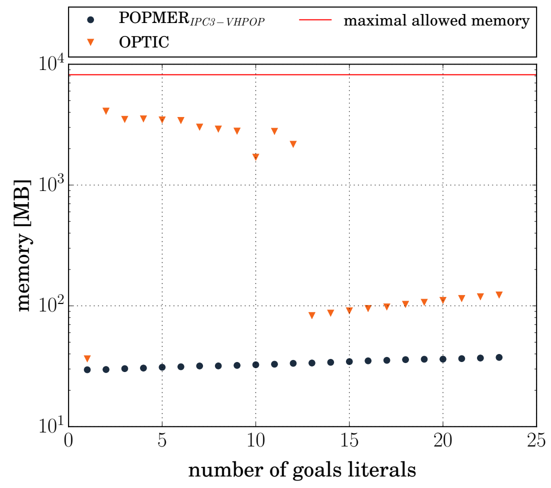
The memory scalability is also affected. Note that problems with 2 - 12 goal literals in Fig. 13.8c where the memory used is higher than in the previous comparisons. However, it may seem that the usage of memory improves for problems with more goals from the RR set (Fig. 13.8d). Nevertheless, recall that we are reporting only on the memory at the end of the run. In these cases, OPTIC has kept improving its solution for problems with 1-8 goal literals, leading to not only a better score than POPMER, but also to a higher amount of used memory. However, for problems with more goal literals, it has not improved as much. Therefore, the memory is relatively smaller for these problems.



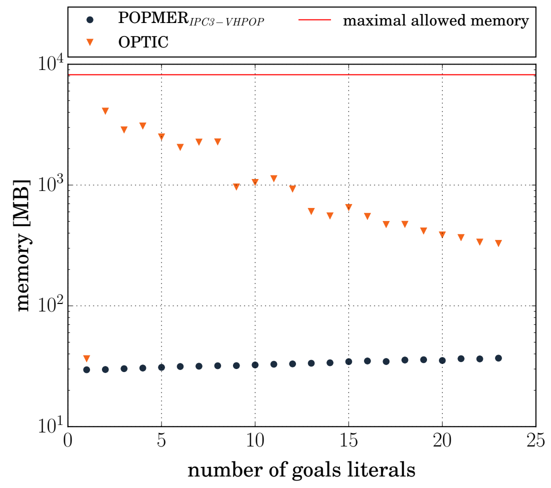
(a) Sequential release dates - *RIPCscore*



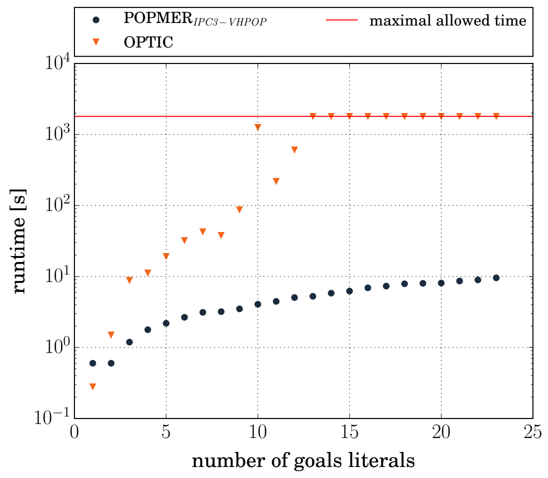
(b) Random release dates - *RIPCscore*



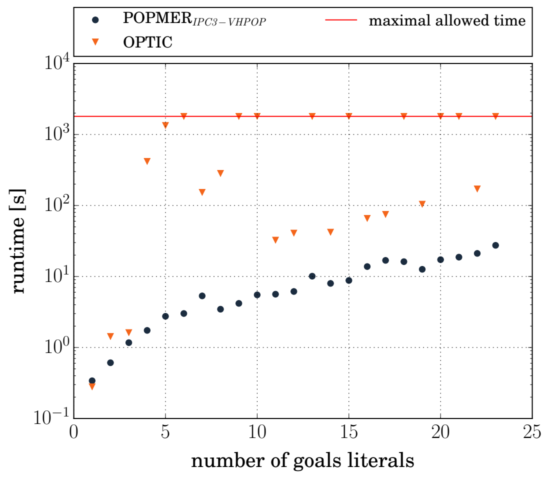
(c) Sequential release dates - memory



(d) Random release dates - memory



(e) Sequential release dates - runtime



(f) Random release dates - runtime

Figure 13.8: The comparison of POPMER$_{\text{IPC3-VHPOP}}$ and OPTIC on the problems with with sequential or random release dates.

13.3.4 Summary

The evaluation on the Driverlog domain with task time windows demonstrated how POPMER$_X$ is able to handle the time windows and outperform OPTIC on problems modelled not only with PDDL 2.2 but also with PDDL 3.1. Not only does POPMER$_{\text{IPC3-VHPOP}}$ have the best coverage on all the tested problems, but also achieves significantly better $RIPCscores$ than OPTIC with an average difference of $36\,\%$ on all the problems. The results also demonstrated that POPMER has deterministic behaviour, which may be beneficial in a real world application to accommodate time for planning. Furthermore, the proposed heuristic drives POPMER to quickly find high quality solutions.

13.4 Conclusion

We have evaluated POPMER$_{\text{IPC3-VHPOP}}$ on the Driverlog domain with durative actions and different settings of time windows. We first conclude on the runtime of POPMER. On the one hand, POPMER$_{\text{IPC3-VHPOP}}$ is significantly slower than the other planners on the problems without time windows; it performs 20 times slower than the wrapped planner IPC3-VHPOP or TFD. On the other hand, POPMER is 10 times faster on problems which have time windows that do not overlap, compare to the previous case. The reason behind this difference is in the need to compute joining plans. In the case without time windows, all input plans can interact because they are not restricted by time windows. Therefore, POPMER computes many joining plans. Moreover, it even generates impossible problems to solve. Even though IPC3-VHPOP recognises such problems quickly, it is a bottleneck for POPMER$_{\text{O-VHPOP}}$, which needs to include a threshold to stop O-VHPOP from running. In the case of non-overlapping time windows, plans do not interact at all. Thus, POPMER does not compute any joining plan.

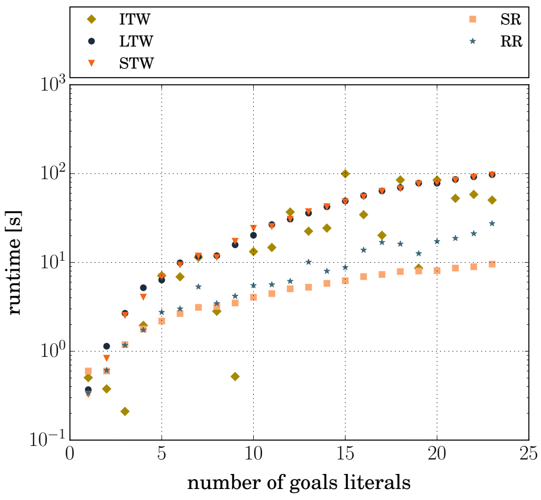


Figure 13.9: The comparison of runtimes of POPMER$_{\text{IPC3-VHPOP}}$ on problems with different time windows.

Fig. 13.9 visualises all runtimes on different problems in a single graph. The runtime on LTW problems is the same as on problems without time windows. The runtime on randomly generated release dates is higher than on those sequentially generated, because this set contains small overlaps between some tasks. Hence, some joining plans are needed. In contrast, the performance on STW and LTW is almost identical because the time windows for all tasks overlap. The only difference is that some problems of STW are unsolvable. Finally, the performance on ITW set depends on the small size of the time windows. For example, for the problem with 9 goal literals, one of the time windows was so small that the plan for the task could not fit in. Hence, POPMER stopped quickly. However, on many problems, POPMER needed to search and backtrack to find that there is no solution. From this comparison, we can conclude that computation of joining plans strongly affects POPMER runtime. We have proposed two implementation changes to limit the computation of the joining plans and to bring the runtime of POPMER closer to the performance on the SR set.

The other studied property was the usage of memory. On all the addressed Driverlog problems, POPMER$_{\text{IPC3-VHPOP}}$ demonstrated the best memory scalability. The worst performance is noticeable on the TMS domain, which is caused by using O-VHPOP as the wrapped planner. However, POPMER performance shows that it significantly improves on the memory usage of its wrapped planner, either O-VHPOP or IPC3-VHPOP.

Finally, $\text{POPMER}_{\text{IPC3-VHPOP}}$ significantly outperforms the other planners used in this comparison. On the domain without time windows, only $\text{DAE}_{\text{YAHSP}}$ scored $5\,\%$ better than POPMER. This planner is unable to handle not only required concurrency, but also time windows. Nevertheless, POPMER and OPTIC handle both of these features, but $\text{POPMER}_{\text{IPC3-VHPOP}}$ is better at $RIPCscore$ by $30\,\%$ on average, compared to OPTIC on the tested problems.

CHAPTER 14

DISCUSSION AND CONCLUSION

The problem addressed in this thesis is motivated by a mobile service robot, which receives tasks to perform from different users. Hence, the robot can plan how to achieve each task separately and then reason about synergies and demands in these plans in order to optimise its overall performance. The synergies and demands arise not only from the different timing characteristics of tasks, but also from the spatial constraints of the environment. Therefore, if the robot can merge activities happening at the same locations, it can optimise its performance. Thus, we have proposed POPMER$_X$, an algorithm that merges partially ordered plans with durative actions, which are constrained not only in space but also in time.

POPMER$_X$ is a complete and sound, forward-chaining algorithm that utilises the same representation as temporally-lifted planners. Additionally, it uses a domain specific preprocessing step, which produces relaxed input plans in order to avoid local optimisation of demands. Furthermore, it utilises a state-of-the-art planner in order to obtain separate plans for each of the relaxed plans. Then, it iteratively reasons with these plans and extracts candidates that are merged into a partially ordered plan. Importantly, the merging process preserves the constraints between actions imposed in the input plans. This is important in order to support required concurrency between actions.

In comparison to existing plan merging techniques, POPMER combines planning with refinements and plan repair. In contrast to plan refinements, it is a forward-search. Hence, it does not need to find all conflicts before hand. It solves them on-the-fly by promoting the conflicting action. Moreover, POPMER considers preconditions and effects of actions in order to optimise its solution. In contrast, plan merging via refinements finds only a valid solution and does not consider its quality. In comparison to plan repair, POPMER interleaves all actions from the input plans, not only their part. To summarise, POPMER's uniqueness is in optimising the merged plan.

We have provided an evaluation of POPMER_X using the VHPOP planner on two benchmark domains – the Driverlog and TMS domain. The Driverlog domain was chosen as it models a similar scenario to the motivation domain (a mobile service robot) and it is constrained by spatial and temporal constraints. The TMS domain requires concurrency between actions in all valid solutions. The evaluation on those domains has shown that $\text{POPMER}_\text{VHPOP}$ is competitive with state-of-the-art temporal planners for the class of problems we are interested in. Furthermore, it has illustrated the flexibility of POPMER, as it can perform well in the two domains we analysed, while the other approaches have issues in at least one of the domains.

However, POPMER_X is built on four restricting assumptions (A1-A4). Hence, it cannot handle some features compared to some other planners. Nevertheless, POPMER_X can be extended in the following ways in order to not require the assumptions:

- Support for multi-agent behaviour (**A1**) can be added, for example, by integrating POPMER_X with an auctioning mechanism or a multi-agent to a single agent translation step as used by MAPL (Brenner, 2003).

- The full observability (**A2**) is unrealistic in many real world scenarios. We propose to deal with partial observability through flexible execution which can be supported by partially ordered plans (POP) for two reasons:

    1. the robot can order different branches of POP based on new observations of its environment;
    2. POP supports generation of temporally flexible plans. This means that actions do not have to be performed at the scheduled time but they can be performed within a time slack, as long as they do not violate deadlines or any of the causal links.

- In order to provide support for resources (**A3**), we could include many existing techniques of resource handling in temporal planners. Importantly, resources will add more demands and synergies that POPMER would need to consider.
- We argue that the fourth assumption (**A4**), that tasks are separable, does not have to be overcome for a real world application. Such property naturally results from the fact that tasks are given by different users. Moreover, tasks, which are not separable, can be handled as one bigger task if needed.

Another aspect to improve is the POPMER heuristic. It would be interesting to explicitly study the influence of the heuristic on POPMER behaviour and compare it with already existing heuristics. Moreover, domain independent relaxation of the individual planning problems should be developed in order to make POPMER automatic on any domain. From the implementation point of view, POPMER could be optimised to achieve better runtime, especially by eliminating the need to compute joining plans often. Furthermore, it could be extended to be an anytime planner in order to overcome that the heuristic drives it into a local optimum. This would be done by not terminating after the first solution is found. Thus, POPMER would be able to explore other choices.

Finally, we conclude this thesis as a whole. Our goal was to study different deliberation techniques, in particular scheduling and temporal planning, and to address how these techniques could be improved in order to handle not only temporal, but also spatial constraints. Our motivation was a scenario from a mobile service robot domain where the robot needs to perform tens of tasks per hour. Therefore, we were also interested in how good scalability of the deliberation techniques can be achieved.

In the first part of the thesis, focused on scheduling, we proposed the pruning scheduler, which heuristically choose an order for tasks with overlapping time windows. We have demonstrated how such pruning has significantly improved the scalability, compared to the scheduler deployed on the CoBots (Coltin et al., 2011). Moreover, our scheduler has been deployed on real robots for several months (Hawes et al., To Appear).

In the second part, we moved beyond the atomic tasks required by scheduling. As a result, we have built on temporal planning in order to plan how to achieve the tasks as well. Our proposed algorithm POPMER$_X$ outperforms state-of-the-art planners on the class of problems we tackle in this thesis. Furthermore, POPMER$_X$ is able to address problems that require release dates and deadlines on tasks. This area of research has not been thoroughly addressed before in state-based temporal planners.

BIBLIOGRAPHY

Achterberg, T. (2009). SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, *1*(1), 1–41. http://mpc.zib.de/index.php/MPC/article/view/4.

Aker, E., Erdogan, A., Erdem, E. & Patoglu, V. (2011, March). Housekeeping with Multiple Autonomous Robots: Representation, Reasoning and Execution. In *Logical Formalizations of Commonsense Reasoning, AAAI Spring Symposium*. Stanford, California,USA.

Allen, J. F. (1983). Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, *26*(11), 832–843.

Allouche, M. K. & Boukhtouta, A. (2010). Multi-agent coordination by temporal plan fusion: Application to combat search and rescue. *Information Fusion*, *11*(3), 220–232. Agent-Based Information Fusion. doi:http://doi.org/10.1016/j.inffus.2009.09.005

Atay, N. (2007). Emergent task allocation for mobile robots. In *Proceedings of Robotics: Science and Systems*.

Bacchus, F. & Ady, M. (2001). Planning with Resources and Concurrency: A Forward Chaining Approach. In B. Nebel (Ed.), *Proceedings of International Joint Conference on Artificial Intelligence - IJCAI 2001* (pp. 417–424). Morgan Kaufmann.

Bäckström, C. & Nebel, B. (1995). Complexity Results for SAS+ Planning. *Computarional Intelligence*, *11*, 625–655.

Barreiro, J., Boyce, M., Do, M., Frank, J., Iatauro, M., Kichkaylo, T., . . . Smith, D. (2012). EUROPA: A Platform for AI Planning, Scheduling, Constraint Programming, and Optimization. In *4th International Competition on Knowledge Engineering for Planning and Scheduling - ICKEPS 2012).*

Beetz, M. & Bennewitz, M. (1998). Planning, scheduling, and plan execution for autonomous robot office couriers. In *Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments, volume Workshop Notes* (pp. 98–02).

Benedictis, R. D. & Cesta, A. (2015, September). New Heuristics for Timeline-Based Planning. In *Proceedings of the 6th Italian Workshop on Planning and Scheduling A workshop of the XIV International Conference of the Italian Association for Artificial Intelligence - AI*IA 2015* (pp. 33–48). Ferrara, Italy.

Benton, J., Coles, A. J. & Coles, A. (2012, June). Temporal Planning with Preferences and Time-Dependent Continuous Costs. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling - ICAPS 2012.* Atibaia, São Paulo, Brazil.

Berry, P. M. (1993). Uncertainty In Scheduling: Probability, Problem Reduction, Abstractions And The User. In *IEEE Colloquium on Advanced Software Technologies for Scheduling, Digest No.*

Bibaï, J., Savéant, P., Schoenauer, M. & Vidal, V. (2010, May). An Evolutionary Metaheuristic Based on State Decomposition for Domain-Independent Satisficing Planning. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling - ICAPS 2010* (pp. 18–25). Toronto, Ontario, Canada.

Bidot, J., Schattenberg, B. & Biundo, S. (2008, September 23). Plan Repair in Hybrid Planning. In A. R. Dengel, K. Berns, T. M. Breuel, F. Bomarius & T. R. Roth-Berghofer (Eds.), *Advances in Artificial Intelligence: 31st Annual German*

Conference on AI - KI 2008 (pp. 169–176). Kaiserslautern, Germany, doi:10.1007/978-3-540-85845-4_21

Bidot, J., Vidal, T., Laborie, P. & Beck, J. (2009). A theoretic and practical framework for scheduling in a stochastic environment. *Journal of Scheduling*, *12*(3), 315–344.

Bit-Monnot, A. (2016). *Temporal and Hierarchical Models for Planning and Acting in Robotics* (Doctoral dissertation, l'Institut National Polytechnique de Toulouse (INP Toulouse)).

Booth, K. E. C., Tran, T. T., Nejat, G. & Beck, J. C. (2016, January). Mixed-Integer and Constraint Programming Techniques for Mobile Robot Task Planning. *IEEE Robotics and Automation Letters*, *1*(1), 500–507. doi:10.1109/LRA.2016.2522096

Brenner, M. (2003). Multiagent Planning with Partially Ordered Temporal Plans. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence - IJCAI 2003* (pp. 1513–1514). Acapulco, Mexico.

Burgard, W., Cremers, A. B., Fox, D., Haenel, D., Lakemeyer, G., Schulz, D., ... Thrun, S. (1998). The Interactive Museum Tour-Guide Robot. In *Proceeding of the Fifteenth National Conference on Artificial Intelligence - AAAI 1998.*

Caldiran, O., Haspalamutgil, K., Ok, A., Palaz, C., Erdem, E. & Patoglu, V. (2009, September). Bridging the Gap between High-Level Reasoning and Low-Level Control. In E. Erdem, F. Lin & T. Schaub (Eds.), *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning - LPNMR 2009* (pp. 342–354). Potsdam, Germany. doi:10.1007/978-3-642-04238-6_29

Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., Carrera, A., ... Carreras, M. (2015, June 7). ROSPlan: Planning in the Robot Operating System. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015* (pp. 333–341). Jerusalem, Israel.

Cesta, A., Cortellessa, G., Fratini, S. & Oddi, A. (2009). MrSPOCK: A long-term planning tool for MARS EXPRESS. In *6th International Workshop on Planning and Scheduling for Space - IWPSS 2009)*. Pasadena, CA.

Cesta, A., Cortellessa, G., Oddi, A., Policella, N. & Susi, A. (2001). A constraint-based architecture for flexible support to activity scheduling. In *AI* IA 2001: Advances in Artificial Intelligence* (pp. 369–381). Springer.

Cesta, A., Cortellessa, G., Rasconi, R., Pecora, F., Scopelliti, M. & Tiberio, L. (2011). Monitoring elderly people with the Robocare Domestic Environment: Interaction synthesis and user evaluation. *Computational Intelligence*, *27*(1), 60–82.

Cesta, A. & Fratini, S. [Simone]. (2008). The Timeline Representation Framework as a Planning and Scheduling Software Development Environment. In *The 27th PLANSIG Workshop of the UK*.

Chen, X., Ji, J., Jiang, J., Jin, G., Wang, F. & Xie, J. (2010). Developing High-level Cognitive Functions for Service Robots. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems - AAMAS 2010* (pp. 989–996). Toronto, Canada: International Foundation for Autonomous Agents and Multiagent Systems.

Chen, X., Jin, G. & Yang, F. (2012). Extending C+ with Composite Actions for Robotic Task Planning. In *Technical Communications of the 28th International Conference on Logic Programming - ICLP 2012* (Vol. 17, pp. 404–414). Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany. doi:10.4230/LIPIcs.ICLP.2012.404

Chien, S., Rabideau, G., Knight, R., Sherwood, R., Engelhardt, B., Mutz, D., ... Tran, D. (2000). ASPEN - Automated Planning and Scheduling for Space Mission Operations. In *in Space Ops*.

Cialdea Mayer, M., Orlandini, A. & Umbrico, A. (2016). Planning and execution with flexible timelines: a formal account. *Acta Informatica*, *53*(6), 649–680. doi:10.1007/s00236-015-0252-z

Coles, A. I., Fox, M., Long, D. & Smith, A. J. (2008, July). Planning with Problems Requiring Temporal Coordination. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence - AAAI 2008*.

Coles, A. J. [A. J.], Coles, A. I., Fox, M. & Long, D. (2010, May). Forward-Chaining Partial-Order Planning. In *The International Conference on Automated Planning and Scheduling - ICAPS 2010)*.

Coles, A. J. [Amanda Jane], Coles, A., Fox, M. & Long, D. (2012). COLIN: Planning with Continuous Linear Numeric Change. *Journal of Artificial Intelligence Research*, *44*.

Coles, A., Fox, M., Long, D. & Smith, A. (2008). Planning with Problems Requiring Temporal Coordination. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence* (pp. 892–897). AAAI'08. Chicago, Illinois: AAAI Press.

Coltin, B. & Veloso, M. (2014). Optimizing for Transfers in a Multi-vehicle Collection and Delivery Problem. In M. Ani Hsieh & G. Chirikjian (Eds.), *Distributed Autonomous Robotic Systems: The 11th International Symposium* (pp. 91–103). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-55146-8_7

Coltin, B., Veloso, M. M. & Ventura, R. (2011). Dynamic User Task Scheduling for Mobile Robots. In *Automated Action Planning for Autonomous Mobile Robots* (Vol. WS-11-09). AAAI Workshops. AAAI.

Cushing, W. A. (2012). *When is Temporal Planning Really Temporal?* (Doctoral dissertation, Arizona State University).

Cushing, W., Kambhampati, S., Mausam & Weld, D. S. (2007, January). When is Temporal Planning Really Temporal? In *Proceedings of the 20th International Joint Conference on Artificial Intelligence - IJCAI 2007* (pp. 1852–1859). Hyderabad, India.

Dang, Q.-V. & Nguyen, L. (2016). A Heuristic Approach to Schedule Mobile Robots in Flexible Manufacturing Environments. *Procedia CIRP*, *40*, 390–395. doi:10.1016/j.procir.2016.01.073

Davenport, J. A. & Beck, J. C. (2000). *A Survey of Techniques for Scheduling with Uncertainty*.

de Weerdt, M. (2003). *Plan Merging in Multi-Agent Systems* (PhD thesis, Delft University of Technology).

Dechter, R., Meiri, I. & Pearl, J. (1991). Temporal constraint networks. *Artificial intelligence*, *49*(1), 61–95.

Do, M. B. & Kambhampati, S. (2011). SAPA: A Multi-objective Metric Temporal Planner. *Journal of Artificial Intelligence Research, 1-40*.

Dvořák, F., Bit-Monnot, A., Ingrand, F. & Ghallab, M. [Malik]. (2014, June). A Flexible ANML Actor and Planner in Robotics. In *ICAPS PlanRob WorkShop*. Portsmouth, USA.

Dvořák, F., Bit-Monnot, A., Ingrand, F. & Ghallab, M. (2014, June). A Flexible ANML Actor and Planner in Robotics. In *ICAPS PlanRob WorkShop*. Portsmouth, USA.

Edelkamp, S. & Hoffmann, J. (2004). *PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition (PDF)*. Albert-Ludwigs-Universität Freiburg, Institut für Informatik.

Ephrati, E. & Rosenschein, J. S. (1993). Multi-Agent Planning as the Process of Merging Distributed Sub-plans. In *In Proceedings of the Twelfth International Workshop on Distributed Artificial Intelligence -DAI 1993* (pp. 115–129).

Erdem, E., Aker, E. & Patoglu, V. (2012). Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intelligent Service Robotics*, *5*(4), 275–291. doi:10.1007/s11370-012-0119-x

Eyerich, P., Mattmüller, R. & Röger, G. (2009). Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning. In *Proceedings of the 19th*

International Conference on Automated Planning and Scheduling -ICAPS 2009. Thessaloniki, Greece.

Fikes, R. E. & Nilsson, N. J. [Nils J.]. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence - IJCAI 1971* (pp. 608–620). London, England: Morgan Kaufmann Publishers Inc.

Foulser, D. E., Li, M. & Yang, Q. (1992). Theory and algorithms for plan merging. *Artificial Intelligence*, *57*(2–3), 143–181.

Fox, M. & Long, D. (2003, December). PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, *20*(1), 61–124.

Fox, M. & Long, D. (2006, October). Modelling Mixed Discrete-continuous Domains for Planning. *Journal of Artificial Intelligence Research*. *27*(1), 235–297.

Frank, J. & Jónsson, A. (2003). Constraint-Based Attribute and Interval Planning. *Constraints*, *8*(4), 339–364. doi:10.1023/A:1025842019552

Fratini, S. [S.], Pecora, F. & Cesta, A. (2008). Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. *Archives of Control Sciences*, *18*(2), 231–271.

Fratini, S. [Simone] & Cesta, A. (2012). The APSI framework: a platform for timeline synthesis. In *Workshop on Planning and Scheduling with Timelines* (pp. 8–15).

Gelfond, M. & Lifschitz, V. (1998). Action Languages. *Electronic Transactions on AI*, *3*.

Gerevini, A. E., Haslum, P., Long, D., Saetti, A. & Dimopoulos, Y. (2009, April). Deterministic Planning in the Fifth International Planning Competition: PDDL3 and Experimental Evaluation of the Planners. *Artificial Intelligence*, *173*(5-6), 619–668.

Gerevini, A. & Long, D. (2005). *Plan constraints and preferences in PDDL3 - the language of the fifth international planning competition*.

Ghallab, M. [M.], Howe, A., Knoblock, C., Mcdermott, D., Ram, A., Veloso, M., ... Wilkins, D. (1998). *PDDL—The Planning Domain Definition Language*. Yale Center for Computational Vision and Control.

Ghallab, M. [Malik] & Laruelle, H. (1994). Representation and Control in IxTeT, a Temporal Planner. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems - AIPS 1994* (pp. 61–67). Chicago, Illinois: AAAI Press.

Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N. & Turner, H. (2004). Nonmonotonic causal theories. *Artificial Intelligence*, *153*(1), 49–104. doi:http://dx.doi.org/10.1016/j.artint.2002.12.001

Griva, I., Nash, S. G. & Sofer, A. (2008). *Linear and Nonlinear Optimization* (2nd). Society for Industrial Mathematics.

Harris, C. & Dearden, R. (2012). Contingency planning for long-duration AUV missions. In *Autonomous Underwater Vehicles (AUV), 2012 IEEE/OES* (pp. 1–6). IEEE.

Hashmi, M. A. & El Fallah Seghrouchni, A. (2010). Merging of Temporal Plans Supported by Plan Repairing. In *Proceedings of the 2010 22Nd IEEE International Conference on Tools with Artificial Intelligence - ICTAI 2010* (pp. 87–94). doi:10.1109/ICTAI.2010.88

Havur, G., Haspalamutgil, K., Palaz, C., Erdem, E. & Patoglu, V. (2013, May). A case study on the Tower of Hanoi challenge: Representation, reasoning and execution. In *IEEE International Conference on Robotics and Automation - ICRA 2013* (pp. 4552–4559). doi:10.1109/ICRA.2013.6631224

Hawes, N., Burbridge, C., Jovan, F., Kunze, L., Lacerda, B., Mudrová, L., ... Hanheide, M. (To Appear). The STRANDS Project: Long-Term Autonomy in Everyday Environments. *IEEE Robotics and Automation Magazine*.

Heinz, S. & Beck, J. C. (2011). Solving Resource Allocation/Scheduling Problems with Constraint Integer Programming. In M. A. Salido, R. Barták & N. Policella (Eds.), *Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems - COPLAS 2011* (pp. 23–30).

Hoffmann, J. (2002, July). Extending FF to Numerical State Variables. In *Proceedings of the 15th European Conference on Artificial Intelligence -ECAI 2002* (pp. 571–575). Lyon, France.

Hoffmann, J. & Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, *14*, 253–302.

Hsu, C. W., Wah, B. W., Huang, R. & Chen, Y. X. (2006, June). Handling Soft Constraints and Preferences in SGPlan. In *Proceedings of ICAPS Workshop on Preferences and Soft Constraints in Planning*.

Kambhampati, S., Knoblock, C. A. & Yang, Q. (1995). Planning as Refinement Search: A Unified Framework for Evaluating Design Tradeoffs in Partial-Order Planning. *Artificial Intelligence*, *76*, 167–238.

Khandelwal, P., Yang, F., Leonetti, M., Lifschitz, V. & Stone, P. (2014). Planning in Action Language BC While Learning Action Costs for Mobile Robots. In *Proceedings of the Twenty-Fourth International Conferenc on International Conference on Automated Planning and Scheduling - ICAPS 2014* (pp. 472–480). Portsmouth, New Hampshire, USA: AAAI Press.

Khandelwal, P., Zhang, S., Sinapov, J., Leonetti, M., Thomason, J., Yang, F., ... Stone, P. (2017). BWIBots: A platform for bridging the gap between AI and human–robot interaction research. *The International Journal of Robotics Research*. doi:10.1177/0278364916688949

Koehler, J. & Hoffmann, J. (2000, June). On Reasonable and Forced Goal Orderings and Their Use in an Agenda-driven Planning Algorithm. *Journal of Artificial Intelligence Research*, *12*(1), 339–386.

Korsah, G. A., Kannan, B., Browning, B., Stentz, A. & Dias, M. B. (2012, May). xBots: An approach to generating and executing optimal multi-robot plans with cross-schedule dependencies. In *2012 IEEE International Conference on Robotics and Automation* (pp. 115–122). doi:10.1109/ICRA.2012.6225234

Krogt, R. v. d. (2005). *Plan Repair in Single-Agent and Multi-Agent Systems* (Doctoral dissertation, TU Delf).

Laborie, P. (2003). Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, *143*(2), 151–188.

Lacerda, B., Parker, D. & Hawes, N. (2014). Optimal and Dynamic Planning for Markov Decision Processes with Co-Safe LTL Specifications. In *Proceeding of International Conference on Intlligent Robots and Systems - IROS 2014*. IEEE.

Lee, J., Lifschitz, V. & Yang, F. (2013). Action Language BC: Preliminary Report. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence - IJCAI 2013* (pp. 983–989). Beijing, China.

Li, H., Sweeney, J., Ramamritham, K., Grupen, R. & Shenoy, P. (2003, May). Real-time support for mobile robotics. In *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium* (pp. 10–18). doi:10.1109/RTTAS.2003.1203032

Long, D. & Fox, M. (2003). Exploiting a Graphplan Framework in Temporal Planning. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling - ICAPS 2003* (pp. 52–61). Trento, Italy: AAAI Press.

Louie, W.-Y. G., Vaquero, T., Nejat, G. & Beck, J. C. (2014). An Autonomous Assistive Robot for Planning, Scheduling and Facilitating Multi-User Activities. In *Proceedings of the IEEE International Conference on Robotics and Automation - ICRA 2014*.

Mali, A. D. (1999, September 8). Plan Merging & Plan Reuse as Satisfiability. In S. Biundo & M. Fox (Eds.), *Recent Advances in AI Planning: 5th European Confer-*

ence on Planning, ECP 1999, (pp. 84–96). Durham, UK, doi:10.1007/10720246_7

Mansouri, M. & Pecora, F. (2014a). More Knowledge on the Table: Planning with Space, Time and Resources for Robots. In *IEEE International Conference on Robotics and Automation - ICRA 2014*.

Mansouri, M. & Pecora, F. (2014b). More knowledge on the table: planning with space, time and resources for robots. In *In Proceedings of IEEE International Conference on Robotics and Automation - ICRA 2014* (pp. 647–654).

McCain, N. & Turner, H. (1997). Causal Theories of Action and Change. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence - AAAI 1997/IAAI 1997* (pp. 460–465). Providence, Rhode Island: AAAI Press.

Mudrová, L. & Hawes, N. (2015, May). Task Scheduling for Mobile Robots Using Interval Algebra. In *2015 IEEE International Conference on Robotics and Automation - ICRA 2015*. Seattle, Washington, USA.

Mudrová, L., Lacerda, B. & Hawes, N. (2015, September). An Integrated Control Framework for Long-term Autonomy in Mobile Service Robots. In *Proceedings of the European Conference on Mobile Robots - ECMR 2015*. Lincoln, UK. doi:10.1109/ECMR.2015.7324192

Mudrová, L., Lacerda, B. & Hawes, N. (2016, August). Partial Order Temporal Plan Merging for Mobile Robot Tasks. In *22nd European Conference on Artificial Intelligence - ECAI 2016* (pp. 1537–1545). The Hague, The Netherlands. doi:10.3233/978-1-61499-672-9-1537

Muscettola, N. (1993). HSTS: Integrating planning and scheduling. In M. Zweben & M. Fox (Eds.), *Intelligent Scheduling* (pp. 169–212). Morgan Kaufmann.

Nau, D., Ghallab, M. & Traverso, P. (2004). *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Nebel, B. & Koehler, J. (1995). Plan Reuse versus Plan Generation: A Theoretical and Empirical Analysis. *Artificial Intelligence*, *76*, 427–454.

Nielsen, I., Dang, Q.-V., Nielsen, P. & Pawlewski, P. (2014). Scheduling of Mobile Robots with Preemptive Tasks. In S. Omatu, H. Bersini, J. M. Corchado, S. Rodríguez, P. Pawlewski & E. Bucciarelli (Eds.), *11th International Conference of Distributed Computing and Artificial Intelligence* (pp. 19–27). Cham: Springer International Publishing. doi:10.1007/978-3-319-07593-8_3

Nilsson, N. J. [N. J.]. (1984, April). *Shakey the robot*. Artificial Intellgence Center Computer Science and Technology Division, SRI.

Pinedo, M. L. (2012). *Scheduling: Theory, Algorithms, and Systems* (4th). Springer Science+Business Media.

Rankooh, M. F. & Ghassem-Sani, G. (2015). ITSAT: An Efficient SAT-Based Temporal Planner. *Journal of Artificial Intelligence Research*, *53*, 541–632. doi:10.1613/jair.4697

Rocco, M. D., Pecora, F. & Saffiotti, A. (2013a). When robots are late: Configuration planning for multiple robots with dynamic goals. In *International Conference on Intelligent Robots and Systems - IROS 2013* (pp. 5915–5922). IEEE.

Rocco, M. D., Pecora, F. & Saffiotti, A. (2013b). When robots are late: Configuration planning for multiple robots with dynamic goals. In *International Conference on Intelligent Robots and Systems - IROS 2013*. IEEE.

Sacerdoti, E. D. (1975). *A Structure for Plans and Behavior* (PhD, Stanford University, Stanford, CA, USA).

Simmons, R. G. & Younes, H. L. S. (2011). VHPOP: Versatile Heuristic Partial Order Planner. *Journal of Artificial Intelligence Research*, *20*.

Smith, D. E. (2016). The Case for Durative Actions: A Commentary on PDDL2.1. Retrieved January 12, 2017, from http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume20/smith03a-html/PDDL-commentary.html

Smith, D. E., Frank, J. & Cushing, W. (2008, September). The ANML Language. In *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS).*

Smith, D. E., Frank, J. & Jónsson, A. K. (2000). Bridging the Gap Between Planning and Scheduling. *KNOWLEDGE ENGINEERING REVIEW*, *15*, 2000.

Smith, D. E. & Weld, D. S. (1999). Temporal Planning with Mutual Exclusion Reasoning. In *Proceedings of the 16th International Joint Conference on Artifical Intelligence -IJCAI 1999* (Vol. 1, pp. 326–333). Stockholm, Sweden.

Stock, S., Mansouri, M., Pecora, F. & Hertzberg, J. (2015). Online Task Merging with a Hierarchical Hybrid Task Planner for Mobile Service Robots. In *International Conference on Intelligent Robots and Systems - IROS 2015*.

Surmann, H. & Morales, A. (2002, August). Scheduling Tasks to a Team of Autonomous Mobile Service Robots in Indoor Enviroments. *Journal of Universal Computer Science*, *8*, 809–833.

Tate, A. (1976, August). *Project Planning Using a Hierarchic Non-linear Planner*. Department of Artificial Intelligence, University of Edinburgh.

Thrun, S., Bennewitz, M., Burgard, W., Cremers, A. B., Dellaert, F., Fox, D., … Schulz, D. (1999). MINERVA: A second-generation museum tour-guide robot. In *Proceedings of IEEE International Conference on Robotics and Automation -ICRA 1999*.

Tonino, H., Bos, A., de Weerdt, M. & Witteveen, C. (2002). Plan coordination by revision in collective agent based systems. *Artificial Intelligence*, *142*(2), 121–145. doi:10.1016/S0004-3702(02)00273-4

Tsamardinos, I., Pollack, M. E. & Horty, J. F. (2000). Merging Plans with Quantitative Temporal Constraints, Temporally Extended Actions, and Conditional Branches. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems - AIPS 2000* (pp. 264–272). Breckenridge, CO, USA.

Veloso, M. M., Biswas, J., Coltin, B., Rosenthal, S., Kollar, T., Mericli, C., … Ventura, R. (2012). CoBots: Collaborative robots servicing multi-floor buildings.

In *International Conference on Intelligent Robots and Systems - IROS 2012* (pp. 5446–5447). IEEE.

Vidal, V. (2004, June). A Lookahead Strategy for Heuristic Search Planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling - ICAPS 2004* (pp. 150–159). Whistler, BC, Canada: AAAI Press.

Vidal, V. (2014, June). YAHSP3 and YAHSP3-MT in the 8th International Planning Competition. In *Proceedings of the 8th International Planning Competition - IPC 2014*. Portsmouth, USA.

Weld, D. S. (1994). An introduction to least commitment planning. *AI magazine*, *15*(4), 27.

Wilkins, D. E. (1988). *Practical Planning: Extending the Classical AI Planning Paradigm*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Yang, Q. (1997). *Intelligent Planning: A Decomposition and Abstraction Based Approach*. London, UK, UK: Springer-Verlag.