UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE FÍSICA

Flexible Plan Execution using Temporal Planning and Probabilistic Models

Tomás Rei dos Santos Ribeiro

Mestrado Integrado em Engenharia Física

Dissertação orientada por:

Professor Rodrigo Martins de Matos Ventura

Professora Guiomar Gaspar de Andrade Evans

2020

Acknowledgements

First, I would like to start by thanking my supervisors, professor Rodrigo Ventura and professor Guiomar Evans, for guiding and supporting me throughout this thesis. I would also like to thank Oscar Lima, Andrea Micheli and Michael Cashmore, for their useful feedback which contributed for improving this thesis. To the Science Faculty of the University of Lisbon, for providing me the perfect environment to grow as an individual. To every professor I've had, they have all contributed to the person I am today and this thesis would never have been possible without their contribution. To the SocRob@Home team, which integrated me beautifully in ISR-Lisboa, provided several unique experiences and taught me how to use ROS. In a more personal note, I would like to thank my girlfriend, Carolina Maia Marques, for always being willing to hear me through my struggles and difficulties, as well as being always ready to help in whatever manner she could. Lastly, I would like to thank my family for raising me from birth till today, contributing in several manners to building the foundations of the person that I am today.

Abstract

Planning is the branch of Artificial Intelligence which concerns the realisation of strategies or action sequences to be executed by intelligent agents. One of the assumptions considered in Classical Planning is the determinism of the world, meaning it is possible to predict with certainty the state resulting from the application of a certain action to another state. However, when the agent executing a plan is a robot interacting with the real world, then the agent is dealing with an uncertain and nondeterministic environment. In the real world, facts can unexpectedly change, actions are not instantaneous and their execution might fail. This makes the planning problem increasingly complex.

This thesis aims to address the problem of executing a temporal plan in a perturbed environment. As a way to solve this problem, an algorithm capable of executing a temporal plan, while resisting unforeseen changes in the world, was developed and tested. The developed algorithm, named Olisipo, takes a probabilistic description of the world and aims to maximise the probability of a successful execution. In addition to this, the Olisipo algorithm acts as an add-on to external planners.

The Olisipo algorithm is an improved version of the algorithm presented in [33]. This algorithm consists of an offline and an online phase. In the offline phase, a totally-ordered plan is relaxed and converted into a partially-ordered plan. In the online phase, Olisipo uses a Branch and Bound search to find the totally-ordered plan with the highest success probability. The success probability of a plan is calculated with a Dynamic Bayesian Network and a probabilistic description of the world. Afterwards, the algorithm tries to execute this plan while it is compatible with the state of the world, allowing for the skipping and repetition of actions, as well as for ignoring irrelevant facts in the world. If the plan stops being compatible with the state of the world, then a new totally-ordered plan is built.

To test the performance of Olisipo, a simulation environment was created with 18 different problems. The Esterel dispatcher, from ROSPlan [8], was extended to act as a comparison term for the Olisipo algorithm. From the obtained results, it was shown that the Olisipo algorithm consistently had a higher probability of successfully solving a problem, performed fewer replans and also executed fewer actions, both for successful and failed executions. Hence, Olisipo offers a substantial improvement in performance for disturbed environments.

Olisipo acts as an add-on to external planners and it has a public source code, so it can be implemented to improve the execution of any other planner.

Keywords: Artificial Intelligence, Robotics, Planning in AI, Temporal Planning, Plan Execution, Bayesian Networks, Dynamic Bayesian Networks, ROS, ROSPlan

Resumo

A inteligência sempre foi um dos fatores mais importantes para a espécia humana, daí nos termos chamado *Homo Sapiens*, que significia "Homem sábio". Desde a Antiguidade Grega que o Homem procura perceber o mundo à sua volta. Este desejo foi evoluindo durante séculos, até que o estudo da lógica matemática levou Alan Turing à Teoria de Computação, que sugere que uma máquina pode simular qualquer ato de dedução lógico através da manipulação de símbolos tão simples como '0' e '1'. Esta ideia levou à criação de uma nova área científica, a Inteligência Artificial (IA), cujo propósito é criar programas capazes de demonstrar comportamento inteligente [42]. Um ramo específico da IA é o Planeamento, que trata a construção e execução de planos. Uma aplicação das tecnologias desenvolvidas em IA é a Robótica, que consiste no ramo de tecnologia responsável pelo desenho, construção, operação e aplicação de robôs, assim como de sistemas de controlo. Contudo, o Planeamento e a Robótica continuam a ser duas áreas científicas distintas, com os seus avanços distintos e conferências próprias. O objetivo desta tese consiste em estabelecer uma ponte entre o Planeamento e a Robótica, ao desenvolver um algoritmo inovador capaz de executar um plano temporal num ambiente perturbado, resistindo a mudanças inesperadas no meio ambiente e ao insucesso de ações.

O algoritmo desenvolvido, denominado Olisipo, é uma evolução de um algoritmo desenvolvido previamente por Oscar Lima, Michael Cashmore, Daniele Magazzeni, Andrea Micheli e Rodrigo Ventura [33]. A versão original do algoritmo possuía uma fase *offline* e uma fase *online*. Este trabalho consistiu em melhorar a fase *online*, com a fase *offline* permanecendo inalterada.

A fase *offline* usa um plano totalmente ordenado $\prod^{tt}$, obtido a partir de um planeador já existente, e relaxa-o para obter um plano parcialmente ordenado $\prod'$. Este plano parcialmente ordenado consiste em: uma rede com nós, correspondentes a todas as ações do plano totalmente ordenado inicial; e relações entre nós, nomeadamente temporais e de interferência.

A fase *online*, que ocorre durante a execução do plano, consiste em utilizar uma procura *Branch and Bound* (B&B) [29] para construir, a partir de $\prod'$, o plano totalmente ordenado com maior probabilidade de execução bem sucedida Γ, tendo em conta o estado atual. A procura B&B utiliza uma Rede de Bayes Dinâmica (RBD), construída iterativamente, para calcular a probabilidade de sucesso de um plano totalmente ordenado e podar a procura. Depois de encontrar Γ, a primeira ação desse plano é executada. Após a execução desta ação, o algoritmo Olisipo executa a última ação, de Γ, que seja compatível com o estado atual. Se nenhuma ação do plano for compatível com o estado atual, então um novo Γ é construído, utilizando $\prod'$ e o estado atual.

Para testar e avaliar o algoritmo Olisipo, desenvolveu-se um ambiente virtual capaz de simular as perturbações do mundo real à execução de um plano. Com este propósito, o ambiente de simulação desenvolvido para o algoritmo original [33] foi extendido. Um código em *Python* foi desenvolvido para perturbar a *Knowledge Base* do ambiente virtual e, o código original escrito em C++, foi alterado para contabilizar a probabilidade de uma ação ser bem sucedida e a probabilidade dos efeitos de uma ação. As perturbações de factos no mundo e de ações ocorrem de forma aleatória, tendo em conta as probabilidades definidas pelo utilizador num ficheiro de texto.

De seguida, estendeu-se o executor Esterel do ROSPlan [8] para servir como termo de comparação ao algoritmo Olisipo. Este simples executor executa um plano totalmente ordenado até a execução de uma ação falhar ou até uma ação deixar de ser aplicável. Quando isto acontece, o executor Esterel constrói um novo plano totalmente ordenado e tenta executá-lo. Este ciclo repete-se até o objetivo ser atingido ou até terem sido efetuados 10 replaneamentos, que foi definido como o número máximo de replaneamentos permitidos para ambos os algoritmos. Utilizou-se o planeador POPF [10] para construir o plano totalmente ordenado do executor Esterel, pois este também foi o planeador utilizado na fase *offline* do algoritmo Olisipo.

Para comparar o algoritmo Olisipo e o executor Esterel, construiu-se um ambiente de simulação de um robô numa fábrica. Neste ambiente, o robô é responsável pela manuntenção de um conjunto de máquinas, de forma a evitar que alguma máquina se avarie e interrompa a produção. Cada máquina possui uma probabilidade diferente de se avariar e pode também receber uma manuntenção aleatória não prevista, atualizando o seu estado. A ação de executar manutenção numa máquina, e de a máquina ser efetivamente mantida, tem também uma probabilidade de sucesso associada. A partir deste ambiente, dois domínios foram construídos, um domínio simples com apenas uma ação não-instanciada possível e outro com duas ações não-instanciadas possíveis. Para o domínio mais simples, foram construídos 10 problemas diferentes. Para o domínio mais complicado, foram construídos 8 problemas. As métricas escolhidas para avaliar o desempenho de cada algoritmo são: a probabilidade de uma execução bem sucedida; o número médio de replaneamentos em execuções bem sucedidas; o número médio de ações em execuções bem sucedidas e falhadas. A análise destas métricas foi feita a partir de 2000 tentativas de resolução de cada algoritmo em cada problema.

Verificou-se que o algoritmo Olisipo supera o executor Esterel em todas as métricas. A partir dos resultados obtidos, é possível afirmar que o algoritmo Olisipo possui uma probabilidade de execução bem sucedida maior ou equivalente ao executor Esterel em todos os problemas testados, sendo que este aumento na probabilidade pode ir até aos 16%. Relativamente a replaneamentos em execuções bem sucedidas, o algoritmo Olisipo apresentou uma mediana de zero em todos os problemas, enquanto o executor Esterel apresentou uma mediana de até 2,2 replaneamentos. Relativamente ao número de ações executadas em execuções bem sucedidas, a mediana da distribuição correspondente ao algoritmo Olisipo foi sempre inferior à mediana correspondente à distribuição do executor Esterel. A melhoria relativa, apresentada pelo algoritmo Olisipo, no número de ações executadas em execuções bem sucedidas variou entre 10% e 52%. No caso de execuções falhadas, o algoritmo Olisipo também possuiu uma mediana inferior ao executor Esterel em todos os problemas, com uma melhoria relativa entre 39% e 58%.

Com base nestes resultados, é possível concluir que o algoritmo Olisipo, relativamente a um simples

executor como o Esterel, possui uma probabilidade de execução bem sucedida mais elevada, precisa de replanear menos vezes e executa menos ações, tanto em execuções bem sucedidas como falhadas.

Estas vantagens do algoritmo Olisipo tornam-se particularmente interessantes em situações em que a execução de uma ação possui um custo ou resulta em consequências negativas.

Considerando o exemplo utilizado anteriormente, de um robô que realiza manuntenção em máquinas numa fábrica, a execução de menos ações contribui para prolongar o tempo de trabalho do robô, uma vez que consumiria menos energia e precisaria de carregar as baterias com menos frequência. Esta vantagem é também útil em situações em que o robô não se encontre facilmente acessível ou em que não pode ser carregado facilmente. Por sua vez, o menor número de replaneamentos possibilita a utilização de um sistema computacional mais simples no robô, contribuindo também para a redução do seu consumo de energia.

O código fonte do trabalho desenvolvido nesta tese é público [1]. Como o algoritmo Olisipo recebe apenas um plano totalmente ordenado e realiza uma execução de forma a maximizar a sua probabilidade de sucesso, é possível aplicar diretamente este algoritmo a outros planeadores, tornando a sua execução mais robusta.

Relativamente ao trabalho futuro a desenvolver, uma possibilidade consiste em, durante a procura B&B, guardar outros planos totalmente ordenados, para além do plano que maximiza a probabilidade de sucesso. Isto pode ser útil pois as perturbações no ambiente podem, inesperadamente, tornar a execução de outro plano na execução com uma maior probabilidade de sucesso. Uma segunda possibilidade consiste em repetir a fase *offline* do Olisipo após um certo números de planos Γ serem consecutivamente construídos ou após não ser possível construir um plano Γ. Uma terceira possibilidade consiste em guardar partes da árvore de procura e RBD, de forma a que, se for necessário procurar por um novo plano Γ e construir uma nova RBD, seja possível utilizar a árvore de procura e RBD anteriores para acelerar e facilitar as suas construções. Uma quarta possibilidade consiste em estudar o impacto da repetição da sub-fase PGen e verificar se podem haver outras soluções mais frutíferas. Uma quinta possibilidade consiste em otimizar o código desenvolvido, utilizando ferramentas como o *Cython* [3] para acelerar a sua execução. Por último, outra possibilidade consiste na implementação e teste do Olisipo num robô real, de forma a avaliar a sua praticabilidade e vantagens de forma mais concreta e relevante. Esta implementação consistiria num trabalho futuro bastante relevante.

Palavras-chave: Inteligência Artificial, Robótica, Planeamento em IA, Planeamento Temporal, Execução de Planos, Redes de Bayes, Redes de Bayes Dinâmicas, ROS, ROSPlan

[1]`https://github.com/TomasRibeiro96/Olisipo-planner`

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

- AI - Artificial Intelligence;
- MCMC - Markov chain Monte Carlo;
- PGen - Plan Generation Sub-Phase;
- PExec - Dynamic Plan Execution Sub-Phase;
- DBN - Dynamic Bayesian Network;
- B&B - Branch and Bound;
- ROS - Robot Operating System;
- STN - Simple Temporal Network;
- SF3 - Simple Factory Robot Domain with 3 machines;
- AF3 - Advanced Factory Robot Domain with 3 machines.

Chapter 1

Introduction

This chapter contextualises and presents the motivation for this thesis, defining the problem it aims to solve. In addition to this, it also outlines the approach taken and states its scientific contributions.

1.1 Motivation and Problem Statement

Planning in Artificial Intelligence concerns the branch of Artificial Intelligence (AI) responsible for the realisation of strategies or action sequences to be executed by intelligent agents. The planning problem consists in using a description of the initial state, of the desired goal and of a set of possible actions to generate a plan which is guaranteed to generate a state which contains the desired goal.

The Classical Planning problem considers a deterministic world, meaning that it is possible to predict, with certainty, the state resulting from the application of one action to another state. The planning problem becomes increasingly complex when the assumption of determinism is dropped. This extension introduces uncertainty in the model of the domain and requires new approaches to planning. Deliberations must take into account that: actions can lead to a set of states; plans are no longer sequences of actions, but conditional plans; solutions may have different strengths. This consideration allows for modelling of uncertainty in the real world, therefore presenting a more realistic description of the world.

The nondeterminism of the real world can affect plan execution in many ways. For example, in the real world actions might fail or have unforeseen consequences. In addition to this, certain facts might be unexpectedly added or removed from the world. The adding and removing of facts from the world might also invalidate the execution of certain actions. Hence, the real world may seem incompatible with the deterministic world of Classical Planning.

The use of a deterministic or a nondeterministic model is a design choice. In some cases, it might be advantageous to simply treat the world as deterministic and, when the execution of an action fails, to recover by replanning. In other cases, a nondeterministic description of the world is essential. Despite nondeterministic models allowing for a more realistic description of the world and to plan for recovery mechanism at design time, they also have clear disadvantages. One disadvantage is the fact that taking into account all the different possible outcomes may be too complicated, both conceptually and computationally.

Another factor which contributes to complicating further more the planning problem consists in temporal planning. In temporal planning problems, actions are not instantaneous and can be executed concurrently. In addition to this, the planning problem might also possess temporal deadlines, time-windows and synchronisations which directly affect the success of the plan.

Problem Statement: The goal of this thesis is to bridge the gap between Nondeterministic Temporal Planning and Robotics, by developing an innovative algorithm for robust plan execution with unexpected observations. This algorithm must allow for the skipping of actions, ignore irrelevant facts in the world and maximise the probability of a successful execution.

1.2 Outline of the approach

The algorithm, over which this thesis is based on, was originally developed by Oscar Lima, Michael Cashmore, Daniele Magazzeni, Andrea Micheli and Rodrigo Ventura [33] and was part of an on going work [32]. The approach taken in [33] consists in first extracting an adaptable partially-ordered plan from a totally-ordered plan, as an offline step. The adaptable partially-ordered plan allows for some causal constraints to be violated, hence enabling a stronger run-time adaptation. After extracting the partially-ordered plan, an online algorithm is used to handle execution. The online algorithm uses an estimation of the probabilities of each planning variable to analyse all of the valid totally-ordered plans induced by the partially-ordered plan. This set of totally-ordered plans is used by a novel action selection policy to choose the next action to execute that maximises the probability of achieving the goal. This execution flow is extremely flexible, since it allows for dynamic re-orderings of the planned set of actions, as well as the skipping of actions that might no longer be needed.

This thesis focuses on extending the online phase of the original algorithm, with the offline phase remaining unchanged.

The extended online phase consists in using a Branch and Bound (B&B) search [29] to build the set of totally-ordered plans. The B&B search uses a plan's success probability as the cost, to prune the search. A plan's success probability is incrementally calculated by resorting to a Dynamic Bayesian Network. In our approach, we assume a probabilistic description of the world. After finding the plan with the highest success probability, that plan is saved and its first action is dispatched for execution. After executing that action, if the same plan remains valid, then it is reused and the lattermost action compatible with the state of the world is dispatched for execution. This avoids the repeated search for new plans, when the already found plan remains valid, and allows for the skipping of actions, making the execution more robust to unpredictable changes in the world.

The source code for the Olisipo algorithm can be found on `https://github.com/TomasRibeiro96/Olisipo-planner`.

From this work, a conference paper was drafted and is to be submitted for ICAPS 2021.

1.3 Contributions

This thesis presents the following scientific contributions:

- A novel algorithm, named Olisipo, which is able to execute a temporal plan in a non-deterministic environment. The Olisipo algorithm can be applied to an already existing planner to improve the probability of a successful execution;
- Results comparing the Olisipo algorithm with a simple executor of a totally-ordered plan;
- A conference paper to be submitted for ICAPS 2021.

Chapter 2

Background and Related Work

This chapter focuses on previous work, accomplished by other researchers, relevant for this thesis. First, it gives a theoretical overview of Artificial Intelligence, Classical Planning, Planning Algorithms, the Quantification of Uncertainty and Bayesian networks. Then, it focuses on the State of the Art for: expressing the domain of a planning problem; building a partially-ordered plan; building a set of totally-ordered plans; obtaining the relevant sub-graph from a Bayesian Network; performing inference on a Bayesian Network. Lastly, it focuses on the tools used for the implementation of the Olisipo algorithm, such as *ROS* and *ROSPlan*.

2.1 Theoretical Background

2.1.1 Artificial Intelligence and Planning in AI

Intelligence has always been one of the most important factors for the human species, hence why we chose to call ourselves *Homo Sapiens*, meaning “man the wise”. Ever since the Ancient Greeks that we have tried to understand how we are able to think, perceive, predict and manipulate the world surrounding us. The field of Artificial Intelligence (AI) aims to go further than this, it aims to not only understand but to also build intelligent entities, such as computer programs and algorithms.

The study of mechanical or “formal” reasoning began with philosophers and mathematicians in Antiquity, namely Aristotles and Chrysippus. These ideas were taught, debated and refined for centuries, until the study of mathematical logic led Alan Turing to the Theory of Computation. Turing suggested that a machine could simulate any conceivable act of mathematical deduction by shuffling symbols as simple as ‘0’ and ‘1’. This idea, along with advancements in neurobiology, information theory and cybernetics, led researchers to the possibility of building an electronic brain. This caused a roaring debate in the scientific community on whether a machine could possess intelligence. Turing proposed to change this question to “whether or not it is possible for machinery to show intelligent behaviour” [42].

In 1943, McCulloch and Pitts completed their formal design for Turing-complete “artificial neurons” [34], which is nowadays recognised as the first work in AI. In 1956, at a workshop at Dartmouth College, the term “Artificial Intelligence” was first coined by John McCarthy [40]. From this definition, Allen Newell (CMU), Herbert Simon (CMU), John McCarthy (MIT), Marvin Minsky (MIT) and Arthur Samuel (IBM) became the founders and leaders of AI research. For the next 20 years, the field would be dominated by these people and their students [40]. They produced computer programs that the press described as “astonishing”. Computers were learning checkers strategies (C. 1954), solving word problems in algebra, proving logical theorems and speaking English[40]. These programs were named *rational agents*.

An **agent** is anything that percepts its environment through sensors and acts on it through actuators, as illustrated in Fig. 2.1. A **rational agent** is an agent that does the right thing, meaning it acts to achieve the best outcome or, in the presence of uncertainty, the best expected outcome [40].

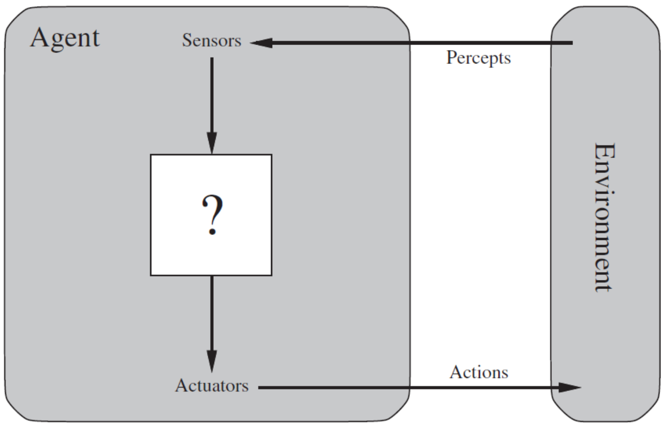


Figure 2.1: An agent interacts with its environment through sensors for perception and actuators for actions [40].

From this definition of rational agent, it is possible to define a class of rational agents: **robots**. According to the *Oxford English Dictionary*, a robot is a machine programmable by a computer and capable of carrying out a complex series of actions automatically.

Deriving from the romantic desire to have a workmate and/or playmate centuries ago, the modern history of robot control starts with the birth of AI in the 50s. At that time, robots were considered to be machines totally under the control of an artificial intelligence, which was able to understand the world and the physics of the body well enough to control the robot. This led to the development of robots capable of performing tasks such as machine welding, painting cars and assembling parts. However, it soon became clear that the complexity of the world far exceeded the internal models used in an AI approach. Even now, 70 years after, behavioural skills of the most advanced robots are far behind those of any insect.

Robotics is the branch of technology that deals with the design, construction, operation and application of robots, as well as computer systems for their control, sensory feedback and information processing.

One branch of AI especially useful for robotics is **AI Planning**, since it concerns the realisation of strategies or action sequences to be executed by intelligent agents. Unlike classical control, the solutions for planning are complex and must be discovered and optimised in multidimensional space. The first example of a planning system being used in robotics was the robot Shakey (Fig. 2.2).

Shakey was a robot developed at the Stanford Research Institute between 1966 and 1972 and it was the first general-purpose mobile robot to be able to reason about its own actions. Shakey consisted of a **goal-based agent**, meaning it percepts the environment's state, it determines what are the effects of each action the robot can perform and checks whether those effects approximate the environment's state to the goal state. Based on this, it would perform the actions that matched the environment's state to the goal.

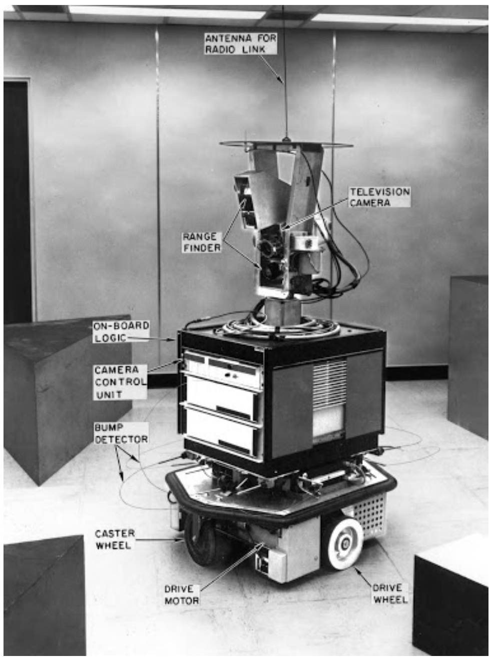


Figure 2.2: A photograph of Shakey taken from the Stanford Research Institute's website: `www.ai.sri.com/shakey`

The Shakey robot was the first to integrate planning in robotics. While other robots needed to be instructed on each individual step of a larger task, Shakey could analyse commands and break them down into basic chucks by itself. The robot made use of *STRIPS*, which was the first major planning system to be developed [16]. The *STRIPS* overall control structure was based on the General Problem Solver [38], a state-space search system that used means-ends analysis. STRIPS' major contribution consisted in setting the standard to what is now known as the "classical" language in planning. STRIPS paved the way for other planning languages, such as the *Problem Domain Definition Language* (*PDDL*).

Although Shakey is a robot that started being developed in the 1960s, there is still a lot of work to be done in integrating Classical Planning in Robotics. The uncertainty of the real world and the fact that actions are not instantaneous make the execution of a Classical Plan in a real-world environment especially complex. Despite several strategies being used to address this problem, such as online planning and nondeterministic domains, no strategy has been found that universally solves this problem for all domains.

2.1.2 Classical Planning

According to [24], a simple planning instance is defined by the domain and problem, $I = (Dom, Prob)$. The domain is a 3-tuple $Dom = (S, A, \gamma)$ or 4-tuple $Dom = (S, A, \gamma, cost)$ [24], where:

- S: finite set of states in which the system may be;
- A: finite set of actions the agent may execute;
- $\gamma : S \times A \rightarrow S$ is a partial function called *prediction function*, which contains the predicted outcome for applying action a in state S;
- $cost : S \times A \rightarrow [0, \infty[$ is a partial function with the same domain as γ that represents the cost of applying action a to state S. If $cost$ is undefined, then $cost(s, a) = 1$ for all applicable actions.

The problem $Prob = (Objs, Init, G)$ is a 3-tuple consisting of the objects in the domain, the initial state and the goal state.

Given a description of the initial state of the world $Init$, of the desired goal G and of a set of possible actions A, **the planning problem is to synthesise a plan that is guaranteed to generate a state which contains the desired goals**. However, the difficulty of planning is dependent on the simplifying assumptions employed. Planning problems can be identified as several classes depending on their properties, such as whether the actions are deterministic, whether the state variables are discrete or continuous, whether the current state can be observed unambiguously, etc.

The simplest possible planning problem, known as the **Classical Planning Problem**, is determined by: a known initial state; instantaneous and deterministic actions; actions that can only be taken one at a time; and one single agent. Since the actions are deterministic and both the initial state and actions' effects are known, it is possible to accurately predict the state of the world after any sequence of actions.

To represent the state of the world, researchers have settled on a factored representation, where the state of the world is represented by a collection of predicates. We are going to use a simplified version of PDDL (*Planning Domain Definition Language*) [22] to show how it describes the four things needed to define a search problem: the initial state; the set of actions available in a state; the results of applying an action to a state; and a goal test.

Each state is represented as a conjunction of fluents that are ground, functionless atoms. For example, $Plane(P1) \wedge Airport(JFK)$ describes a state where *P1* is a plane and *JFK* is an airport. In PDDL, there are two assumptions utilised:

- **Closed world assumption**: any fluents not mentioned are false. In the previous example, since *Car(P1)* was not mentioned, it means that *Car(P1) = False*.

- **Unique names assumption**: all objects have different names and each name is only linked to one object. This implies that *JFK* and, for example, *SFO* are different objects.

Actions can be defined by its preconditions and effects. For an action to be applied, the state of the world has to satisfy the action's preconditions and, after the action is applied, the effects describe the changes in the world. An action should be concise, meaning it should only mention what changed in the world. Let us now consider the following action schema [40]:

$$\begin{aligned}&Action(\ Fly(p, from, to),\\&\quad PRECOND : At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)\\&\quad EFFECT : \neg At(p, from) \land At(p, to)\ )\end{aligned}$$

This schema represents the action $Fly(p, from, to)$ and it is said to be an uninstantiated action, meaning it is only defined by variables, such as p, $from$ and to. These variables can be substituted by constants of the world and give birth to a grounded action:

$$\begin{aligned}&Action(\ Fly(P1, JFK, SFO),\\&\quad PRECOND : At(P1, JFK) \land Plane(P1) \land Airport(JFK) \land Airport(SFO)\\&\quad EFFECT : \neg At(P1, JFK) \land At(P1, SFO)\ )\end{aligned}$$

This action means that if $P1$ is a plane, if JFK and SFO are both airports and the plane $P1$ is at JFK, then it is possible to apply the action $Fly(P1, JFK, SFO)$. This action will have as effects to remove the fluent $At(P1, JFK)$ from the state and to add the fluent $At(P1, SFO)$ to the state. Stating all this in clear English, flying the plane $P1$ from JFK to SFO will make the plane $P1$ stop being at JFK and start being at SFO.

If the state of the world satisfies the preconditions of a certain action, then that action is **applicable**.

Now, to better understand the complete PDDL description of a planning problem, we present below a complete PDDL description of an air cargo transportation planning problem [40]:

$$\begin{aligned}&Init(\ At(C1, SFO) \land At(C2, JFK) \land At(P1, SFO) \land Cargo(C1) \land Cargo(C2)\\&\quad \land Plane(P1) \land Plane(P2) \land Airport(JFK) \land Airport(SFO)\ )\end{aligned}$$

$$Goal(\ At(C1, JFK) \land At(C2, SFO)\ )$$

$$\begin{aligned}&Action(\ Load(c, p, a),\\&\quad PRECOND : At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)\\&\quad EFFECT : \neg At(c, a) \land In(c, p)\ )\end{aligned}$$

$$\begin{aligned}&Action(\ Unload(c, p, a),\\&\quad PRECOND : In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)\\&\quad EFFECT : At(c, a) \land \neg In(c, p)\ )\end{aligned}$$

$$\begin{aligned}&Action(\ Fly(p, from, to),\\&\quad PRECOND : At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)\end{aligned}$$

$EFFECT : \neg At(p, from) \wedge At(p, to)$)

As it can be seen, the PDDL description of the problem needs to take the initial state of the world, the goal state and the uninstantiated actions of the domain. From these, the planner can check which grounded actions are applicable to the initial state and search for the sequence of grounded actions which adds each fact of the goal to the state. When all of the facts from the goal are in the state, then the goal state has been reached.

2.1.3 Planning Algorithms

Based on the Domain and Problem definitions mentioned in the previous section, how can one come up with a plan that achieves the goal? For that purpose, state-space search algorithms are used. There is an endless number of different search algorithms and variations of search algorithms, hence the choice of which one to use depends on the needs and requirements of the system. Does the algorithm always reach a solution? Is the solution reached the optimal solution? How fast is the algorithm in reaching a solution? Is computer memory a limitation? What is the size of the state-space? What is its branching factor? The answer to these questions dictates which search algorithm may be best for the task at hand.

Given the number of different search algorithms, it is only possible for us to mention a few here, hence we will only focus on some of the main ones. However, it is worth noting that the state of the art is the A* search algorithm with domain independent heuristics [45]. The A* search algorithm is a forward state-space search.

2.1.3.1 Forward State-Space Search

Forward search consists in starting from the initial state and exploring possible future states in search for the goal. In Fig. 2.3a), you can see a forward state-space search of the air cargo transportation planning problem mentioned before.

Some of the main algorithms in forward search are:

- **Breadth-first search:** explores all possible states in the world, layer by layer, until reaching a solution. This algorithm will reach the optimal solution, if the path cost is a non-decreasing function of depth, but it is particularly heavy in terms of memory and time requirements [36][30];

- **Depth-first search:** explores one possible path in the state-space until finding a solution or reaching the end of that path. If a solution is not found, it continues to explore a different path. This algorithm may not reach the optimal solution and it may not reach a solution if the path it is exploring is infinite. However, it may be able to reach a solution must faster than breadth-first;

- **A* search:** explores the state that minimises the function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost function, which describes the cost of reaching a certain state, and $h(n)$ is the heuristic function, which evaluates how close each state is to the goal [25];

- **Branch and Bound:** explores the search tree with breadth-first, depth-first or some other algorithm, but keeps track of the cost $g(n)$ of the best solution found so far. While exploring the search space, if a node has a cost $g(n)$ lower than that of the best solution found so far, it prunes the search [29].

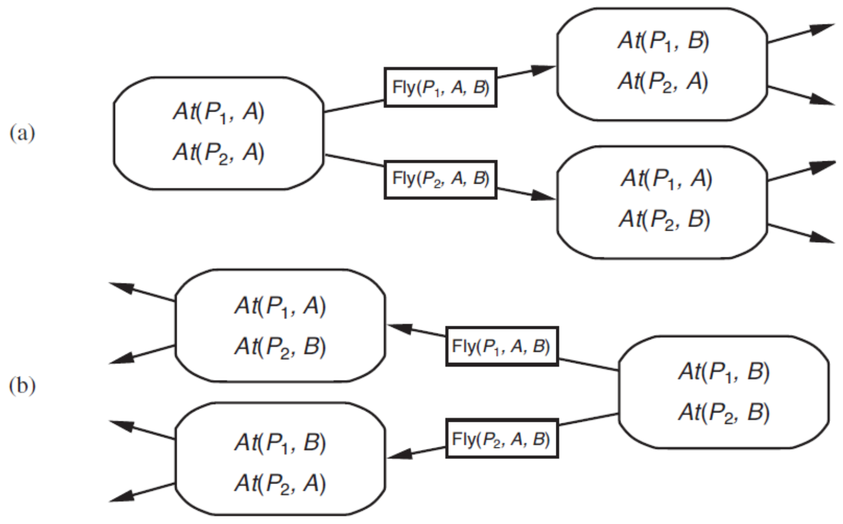


Figure 2.3: (a) Forward search through the space of states, starting in the initial state and using the problem's actions to search forward for the goal. (b) Backward search through sets of relevant states, starting at the goal state and using the inverse of actions to search backwards for the initial state [40].

2.1.3.2 Backward Relevant-States Search

This type of algorithms consist in exploring the relevant-states between the solution and the initial state. They start by taking the solution of the problem, which would be the last state to be explored in Forward search, and backtrack until reaching the initial state. This type of algorithms are called relevant-states search because they only consider actions which are relevant to the goal. Fig. 2.3b) displays the functioning of a backward search.

In general, backward search can only be performed when it is known how to regress from a state to the predecessor state. However, PDDL representation was designed to make it easy to regress actions. Hence, if a domain can be expressed in PDDL, it is viable for backward search.

The algorithms presented in the previous section can also be used in backwards search, but with the search being for the initial state instead of the goal.

2.1.3.3 The GRAPHPLAN Algorithm

This algorithm consists in building a planning graph for the problem at hand. A planning graph, which can be visualised in Figure 2.4, is a directed graph organised into layers, where each layer i represents all the facts that could hold at time i. The nodes in the planning graph represent actions and facts of the world. The edges from facts to actions represent the conditions of an action, the edges from actions to facts represent the effects of an action and the edges from facts to facts represent the persistence of those facts. In addition to nodes and edges, one important addition of the GRAPHPLAN algorithm is the presence of mutual exclusive (mutex) links between fact nodes and between action nodes.

Mutex links between actions represent actions which have inconsistent effects, that interfere with each other or that have compeeting needs. Mutex links between fact nodes represent nodes in the same layer which are the negation of one another or nodes which all ways of reaching them both are mutually exclusive.

The GRAPHPLAN algorithm [5] consists in incrementally adding layers to the planning graph. As soon as all the goals show up as non-mutually-exclusive in the graph, meaning that the path to reach each fact in the goal can coexist with the paths to reach other facts in the goals, it stops expanding and a solution was found .

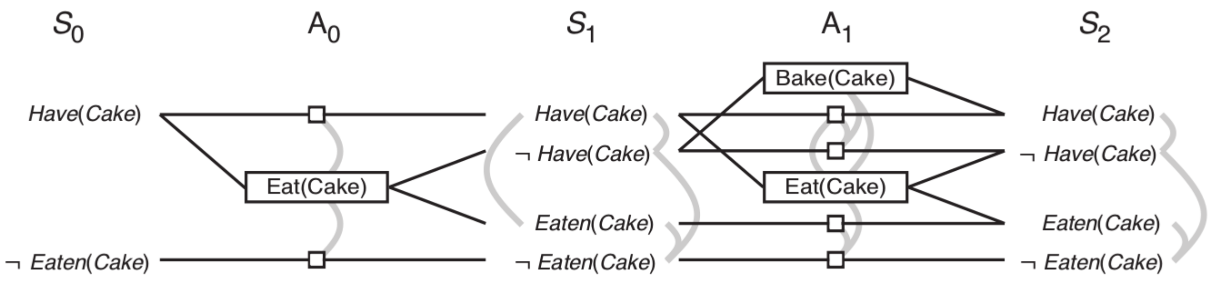


Figure 2.4: The planning graph for the "have cake and eat cake too" problem up to level S_2. Straight lines indicate preconditions and effects. Gray lines indicate mutex links. Not all mutex links are showed, because the graph would be too cluttered [40].

2.1.3.4 The SATPLAN Algorithm

A natural approach to classical AI planning is to treat it as a logical deduction problem. This approach led to the creation of the SATPLAN algorithm. SATPLAN consists on solving planning problems as Boolean satisfiability problems. By translating a PDDL description to propositional logic, it is then possible to solve it by using methods for establishing satisfiability, such as the DPLL algorithm [12] or WalkSAT [41].

2.1.4 Quantifying Uncertainty

To express uncertainty in the world, probabilistic models are used. A fully specified probability model associates a numerical value $p \in [0, 1]$ to each possible event in the world. Probabilities can be split into unconditional or conditional probabilities, which refers to degrees of belief in propositions in the absence or presence of any other information, respectively.

Probabilities such as $P(a) = 0.7$ are called unconditional probabilities, because they do not depend on any other information than the event itself. In the case of $P(a) = 0.7$, the probability of event a only depends on a. Probabilities such as $P(a|b) = 0.6$ are called conditional probabilities, meaning that the probability of the event depends on other information beyond the event. In the case of $P(a|b) = 0.6$, the probability of event a depends on event b.

In addition to calculating the probability of events, it is also possible to calculate the joint probability of different events. For example, $P(a, b)$ represents the probability of event a and event b. The joint probability of events can also be conditional $P(a, b|c)$ or unconditional $P(a, b)$.

The probability of an event can be calculated by summing out the joint probability of all other events. In the situation, of events a and b, this is represented by Equation 2.1.

$$P(a) = \sum_b P(a, b) \tag{2.1}$$

The conditional probability of an event can be calculated with Equation 2.2. The second equality of Equation 2.2 comes from applying Equation 2.1 to its denominator.

$$P(b|a) = \frac{P(a, b)}{P(a)} = \frac{P(a, b)}{\sum_b P(b, a)} \tag{2.2}$$

By manipulating Equation 2.2, it is also possible to obtain Equation 2.3, which is known as the Chain Rule. The Chain Rule can be used to calculate the joint probability of events. A more general version of the Chain Rule is presented in Equation 2.4.

$$P(a, b) = P(a|b) \cdot P(b) \tag{2.3}$$

$$P(x_1, x_2, ..., x_n) = \prod_{i=1}^{n} P(x_i|parents(X_i)) \tag{2.4}$$

Finally, by merging Equations 2.2 and 2.3, it is possible to obtain Equation 2.5, which is known as the Bayes rule.

$$P(b|a) = \frac{P(a|b) \cdot P(b)}{P(a)} \tag{2.5}$$

2.1.5 Bayesian Networks

A Bayesian network is a data structure which represents the statistical dependencies among variables. It is represented by a directed acyclic graph in which each node is annotated with quantitative probability information. The full specification of the Bayesian network is as follows:

- Each node corresponds to a random variable, which may be discrete or continuous;
- Nodes are connected by a set of directed links or arrows. If there is an arrow from node X to node Y, then node X is said to be a parent of Y;
- Each node X_i has a conditional probability distribution $P(X_i|Parents(X_i))$ that quantifies the effect of the parents on the node.

An arrow in a Bayesian network means that one node has direct influence on another. This suggests that causes should be the parents of effects. Once the topology of the Bayesian network is laid out, one only needs to specify the Conditional Probability Distribution (CPD) for each variable, given its parents.

Fig. 2.5 displays a Bayesian network, taken from [40]. Let us suppose you have a new burglar alarm installed at your home. It is fairly reliable at detecting burglary, but it will sometimes respond to minor earthquakes. You also have two neighbours, John and Mary, who have promised to call you at work when they hear the alarm. John nearly always calls when he hears the alarm, but sometimes confuses the telephone ringing with the alarm and then calls too. Whereas Mary likes loud music and misses the alarm more often. All these factors are represented in the Bayesian network of Fig. 2.5, along with the CPD of each node. Given the evidence of who has or has not called, it is possible to estimate the probability of burglary.

In Fig. 2.5, $Burglary$ is a parent of $Alarm$ and is independent of $Earthquake$. The independence between $Burglary$ and $Earthquake$ means that the probability of occurring a burglary is not affected by the occurrence of an earthquake, meaning $P(Burglary|Earthquake) = P(Burglary)$.

To better understand the CPDs, let us examine the CPD of the node $Alarm$. In this CPD, it is stated that the probability of the alarm ringing knowing there was an earthquake and burglary is 95%, meaning $P(A|B,E) = 0.95$. In general, a table for a Boolean variable with k Boolean parents contains 2^k independently specifiable probabilities.

With the CPDs and Equations 2.1, 2.2, 2.3, 2.4 and 2.5, it is possible to calculate any event from the Bayesian Network.

2.2 State Of The Art

The robust execution of plans is a heavily researched topic in the literature. We will first focus on the state of the art for defining a planning domain model. Then, we will look into the state of the art for building a partially-ordered plan. Afterwards, we will review the state of the art for building a set of totally-ordered

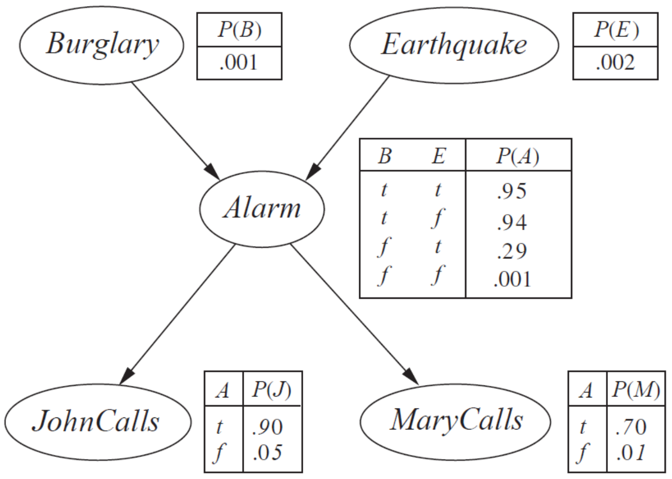


Figure 2.5: A Bayesian network showing the topology and the conditional probability distributions (CPDs). In the CPDs, the letters B, E, A, J and M stand for $Burglary$, $Earthquake$, $Alarm$, $JohnCalls$ and $MaryCalls$, respectively [40].

plans. Subsequently, we will look into the state of the art for obtaining the relevant sub-graph from a Bayesian Netwok. Lastly, we will consider the state of the art for performing inference on a Bayesian Network.

2.2.1 Defining a Temporal Planning Domain Model

The Planning Domain Definition Language (PDDL) is an action-centred language which has become the community standard for the representation and exchange of planning domain models. Drew McDermott and his colleagues were inspired by the STRIPS [16] formulations of planning domain models and first developed PDDL in 1998 [22], as an attempt to create a community standard language which would facilitate the sharing and comparison of different planners. Without PDDL, it would be very hard to compare results of different researchers, since they would be using different tools with different impacts on performance. PDDL was developed for the 1998 International Planning Competition (IPC). Ever since, there have been several editions of the IPC, more specifically in 2000, 2002, 2004, 2006, 2008, 2011, 2014 and 2018, which have resulted in PDDL 1.2 [22], PDDL 2.1 [17], PDDL 2.2 [15], PDDL 3.0 [21] and PDDL 3.1 [28]. These competitions act as a motivator to the research community and PDDL has been incrementally developed for the increasing complexity of the competitions. The PDDL capabilities used in this thesis, such as durative actions, were introduced in PDDL 2.1 [17]. PDDL 2.1 distinguished itself for incorporating temporal planning domains in PDDL.

In PDDL 2.1, time is regarded as point-based rather than interval-based, meaning periods of activity are seen as intervals of state separated by time points, at which state-changing activities occur. To better understand this, let us consider the PDDL description of the durative action *load-truck*, presented below.

```
(: durative-action   load-truck
    : parameters
        (?t − truck)
        (?l − location)
        (?o − cargo)
        (?c − crane)
    : duration (= ?duration 5)
    : condition (and
        (at start (at ?t ?l))
        (at start (at ?o ?l))
        (at start (empty ?c)
        (over all (at ?t ?l))
        (at end (holding ?c ?o))
    : effect (and
        (at end (in ?o ?t))
        (at start (holding ?c ?o))
        (at start (not (at ?o ?l)))
        (at end (not (holding ?c ?o))))
```

First, the parameter $durative\text{-}action$ indicates the nature of the action to the PDDL interpreter. Then, the action's parameters (arguments) are defined. This action receives one parameter of type $truck$, another of type $location$, another of type $cargo$ and another of type $crane$. Then, it is necessary to define the duration of the action, which in this example corresponds to 5 time instants. Regarding the conditions, they are represented as a conjunction of *at start*, *over all* and *at end* fluents. For the action to be applicable, the state of the world needs to satisfy all of the *at start* conditions. In between the action's beginning and end, the environment needs to satisfy the *over all* conditions. When the action is ending, the world needs to satisfy the *at end* conditions. If any of these conditions are not met, the action stops being applicable and is interrupted. In the same way, there are also *at start* and *at end* effects. *At start* effects are applied immediately after the beginning of the action. *At end* effects are only applied after the action is completed. It is necessary for all conditions and effects of durative actions to be temporally annotated, either with an *at start*, *over all* or *at end*.

Formulating this definition of durative action in a more formal manner, a **durative action** a is a tuple $\langle pre(a), eff(a), dur(a)\rangle$ where $pre(a)$ is a set of conditions partitioned into *at start*, *over all* and *at end* conditions; $eff(a)$ is the set of action effects; and $dur(a)$ is the duration constraint. An **instantaneous action** is a tuple $\langle pre(a), eff(a)\rangle$ where $pre(a)$ is the set of preconditions and $eff(a)$ is the set of action effects.

With the introduction of time in domain modelling, the door was opened for concurrent activity to occur in a plan. Before PDDL 2.1, plans were interpreted as sequential, with one action occurring after another. Now, different actions can occur at the same time, therefore leading to increased complexity. In PDDL 2.1 planning problems, the solution can depend on exploiting concurrency correctly.

For a plan to be considered valid, no logical condition can be asserted and negated at the same time. A further constraint is imposed stating that no logical condition can both be required to hold and be asserted at the same instant. A rule named **no moving targets** is adopted, which means that no two actions can simultaneously make use of a value if one of the two is accessing the value to update it.

In order to implement the mutual exclusion relation, PDDL 2.1 requires non-zero-separation between mutually exclusive action end points. When end points are non-conflicting, they can be treated as if it was possible to execute them simultaneously, even though precise synchronicity is not possible in the real world. However, when end points are mutually exclusive, the planner should buffer the co-occurrence of these points by explicitly separating them.

From this, it is possible to formalise a definition of a planning problem and plan using PDDL 2.1 formalism [17].

A **planning problem** is a tuple $\langle P, V, A, I, G \rangle$ where P is a set of propositions; V is a set of real variables; A is a set of durative and instantaneous actions; I is the total function describing the initial state of the propositions and real variables; G is the function indicating the goal condition.

A **plan** $\prod$ for a planning problem $\langle P, V, A, I, G \rangle$ is the graph $\langle N, C \rangle$ where each node $n \in N$ represents the plan start, an instantaneous action, or the start or end of a durative action; and each edge $c \in C$ represents a temporal relation: $x < time(n_1) - time(n_2) < y$ for $n_1, n_2 \in N$ and $x, y \in \mathbb{R}$. Each edge c is labelled as either **causal**, **interference** or **action duration**. Action durations edges express the temporal constraints between the start and end of durative actions. Causal edges express temporal relationships inferred from the causal support between actions. Similarly, interference edges express the temporal relationships inferred from the interference between actions.

A plan is **totally-ordered** if there exists only one total ordering of nodes that can satisfy all of the temporal relations.

2.2.2 Building a Partially-Ordered Plan

One approach for building a partially-ordered plan, taken by some authors, consists in increasing the flexibility of temporal plans to cope with more situations at runtime [6] [14] [19]. In [6], this is achieved according to two criteria: make a plan less constrained and minimise parallel execution time. In [14], it is achieved by generating a partially-ordered plan that uses the same actions as the totally-ordered plan from which it is generated. The authors developed a Constraint Satisfaction Optimisation Problem (CSOP) encoding for converting a position constrained plan in metric/temporal domains into an order-constrained plan.

Another approach consists in synthesising correct-by-construction flexible plans [23] [18] [1] [43]. In [23] the authors describe a planning system called IxTeT, which is a task-level planner for a robot. In IxTeT, time representation takes into account predicted forthcoming events and allows planning in a dynamic world. Knowledge representation has: multi-valued domain attributes temporally qualified into instantaneous events and persistent assertions; an ontology which classifies attributes; hierarchical planning operators; and constraints on domain variables handled specifically. Preprocessing techniques are used at compile-time to check planning operators and input scenarios for consistency and to translate them into more explicit and efficient representations. Causal links organise the search in the space of partial plans. In [18] a paradigm for representing and reasoning about plans is presented. The paradigm enables the description of planning domains with time, resources, concurrent activities, mutual exclusions, disjunctive preconditions and conditional effects.

In the offline phase of Olisipo, a fundamental constraint which was at the base of these previous works is relaxed: the causal structure of the plan is relaxed by discarding causal constraints, in order to allow for more run-time adaptability. The obtained plan admits executions that are invalid for the planning model, so a runtime action selection policy is employed that dynamically selects actions that are causally-valid and are likely to reach the goal. This moves the causal reasoning from the offline phase to the online phase [33].

2.2.3 Building the Set of Totally-Ordered Plans

In what concerns the build of the set of totally-ordered plans during the online phase, we take an approach similar to [31] [13]. The authors of [31] show that a Temporal Plan Network Under Uncertainty is an encoding of a set of many different candidate STN [13] sub-plans. They define a correct execution as an ordering of activities that is causally complete and temporally consistent with respect to the STN. In our approach, we solve the problem of generating the complete set of valid executions for an adaptable partially-ordered plan. A valid execution is a correct execution which also achieves the goal [33].

The same problem is addressed in [26], where the authors select an execution of a Temporal Plan Network compiled from RMPL. In [9], the authors synthesise a dynamically controllable strategy for a disjunctive temporal network with uncertainty. In the solution implemented in this thesis, a similar problem is tackled, where we account for the selected execution's probability of success and do not consider decision nodes. This allows the plan execution to adapt to some unexpected observations. In addition to this, the Branch and Bound search [29] was used to prune the search for the totally-ordered plans.

2.2.4 Obtaining the Relevant Sub-Graph from a Bayesian Network

Simplifying a Bayesian Network is a heavily researched topic. Works on this topic go as back as 1990 [2]. In [2], the developed method resembles the pruning rules utilised in this thesis, with the simplification rules being: 1. remove all nodes that are d-separated from Q by K; 2. remove barren nodes until a node in Q or K is found; 3. remove all edges that are not incident on two nodes. Q is the set of nodes whose values we are interested in and K is the set of nodes with known values.

It is important to note that rule 2 from the simplification rules of [2] is exactly the same as rule 1 of the pruning applied in this thesis. D-separation is an useful concept for the simplification of a Bayesian Network. However, it was not used in this thesis because the constructed Bayesian Network possesses a simple and predetermined structure. Hence, it allows for the derivation of 3 simple and easily implemented rules for its simplification.

A different approach is employed in [35], where evidence sensitivity analysis and diagnostic test selection are used to identify the sets of variables on which computational efforts can focus. Then, this set is related to previously established relevance sets and their computation is compared to these sets. Therefore, an overall picture of the relevance of various variable sets for performing inference and analysis is obtained.

2.2.5 Performing Inference on a Bayesian Network

There are two basic algorithms to perform exact inference on a Bayesian Network: *enumeration* and *variable elimination*.

Inference by **enumeration** consists in summing out the hidden variables in a given query and using the semantics of the Bayesian Network to get an expression in terms of the CPD entries. In the worst case scenario, inference by enumeration may imply summing out almost all variables. A possible implementation of the enumeration algorithm can be found on Figure 14.9 of [40]. However, inference by enumeration can be described as performing inference by brute force and it can have a time complexity of $O(n2^n)$, with n being the number of Boolean variables in the network. Hence, there are other more efficient and less complex methods which can be used to perform inference.

One method, which is a direct improvement from inference by enumeration, is inference by **variable elimination**. This improvement consists in using dynamic programming to save the results of calculations for later use. Intermediate results are stored and summations over each variable are done only for those portions of the expression that depend on the variable. An implementation of this algorithm can be found on Figure 14.11 of [40].

An alternative approach to the variable elimination algorithm is performing inference by **conditioning**. The conditioning algorithm is based on the fact that observing the value of certain variables can simplify the variable elimination algorithm process. When a variable is not observed, we can use a case analysis to enumerate its possible values, perform variable elimination computation, and then aggregate the results for the different values. Inference by conditioning does not offer any benefit over the variable elimination algorithm, in terms of number of operations. However, it offers a continuum of time-space trade-offs, which can be extremely important in cases where the factors created by variable elimination are too big to fit in main memory.

The variable elimination algorithm is simple and efficient for answering individual queries. However, it can be less efficient in the calculation of the posterior probabilities for all the variables in the network. To solve this problem, **clustering** algorithms can be used. For this reason, clustering algorithms are widely used in commercial Bayesian network tools [40]. The basic idea of clustering consists in joining individual nodes of the network to form cluster nodes, in a way that the resulting network is a polytree. Once the network is in polytree form, a special-purpose inference algorithm can be used to determine the posterior probabilities for all the non-evidence nodes in the network.

The inference methods presented so far consist in exact inference methods, meaning they calculate the exact value of the probability to be determined. The complexity of exact inference depends strongly on the structure of the network. For singly connected networks, the time and space complexity of exact inference is linear in the size of the networks. For multiply connected networks, the variable elimination algorithm can have exponential time and space complexity in the worst case scenario[40]. In this type of networks, approximate inference is used.

One class of approximate inference methods are randomised sampling algorithms, also called **Monte Carlo algorithms**. Monte Carlo algorithms with sampling applied to the computation of posterior probabilities have two families of algorithms: direct sampling and Markov chain sampling.

Direct sampling consists in generating events from a network that has no evidence associated with it. Each variable is sampled in turn. The probability distribution from which the value is sampled is conditioned on the values already assigned to the variable's parents. In the case of hard-to-sample distributions, **rejection sampling** [7] [37] [4] may be used to produce samples. **Likelihood weighting** is another method to generate samples in hard-to-sample distributions, which avoids the inefficiency of rejection sampling by generating only events that are consistent with the evidence.

Markov chain Monte Carlo (MCMC) algorithms generate each sample by making a random change to the preceding state. It is helpful to think of an MCMC algorithm as being in a particular current state, specifying a value for every variable, and generating a next state by making random changes to the current state. A particular form of MCMC is **Gibbs sampling** [20], which consists in starting with an arbitrary state (with the evidence variables fixed at their observed values) and generates a next state by randomly sampling a value for one of the non-evidence variables. Hence, the algorithm wanders randomly around the state space, flipping one variable at a time, but keeping the evidence variables fixed.

Another class of approximate inference methods consist in constructing an approximation to the target distribution P_ϕ. This approximation takes a simpler form that allows for inference. First, a target class $\mathcal{Q}$ of "easy" distributions Q is defined, then the instance which is the best approximation to the target distribution is searched for in that class. Queries can then be answered using inference Q rather than on P_ϕ. This approach reformulates the inference task as one of optimising an objective function over the class $\mathcal{Q}$. This problem falls into the category of *constrained optimisation*. Such problems can be solved by a variety of different methods, hence opening the door to the application of a range of techniques developed in the literature. Currently, the technique most often used in the setting of graphical models is one based on the use of *Lagrange multipliers* [27].

The exact inference algorithm developed in this thesis utilises the structure of the constructed Bayesian Network, and the fact that it has been simplified by the 3 pruning rules, to facilitate the calculation. In addition to this, since the calculation is performed incrementally, it saves the result of previous calculations to reduce its complexity. The developed inference algorithm was tested by comparing its results with the Python implementation of the variable elimination algorithm from Figure 14.11 of [40], which can be found on `https://github.com/aimacode/aima-python/blob/master/probability.py`.

2.3 Low-level Implementation Tools

In this thesis, two low-level implementations tools were used. ROS Kinetic[1] was used for communication between different pieces of code running separately. ROSPlan [8] was used to parse the PDDL domain and problem files, as well as convert to them to ROS messages.

2.3.1 Robot Operating System (ROS)

The Robot Operating System (ROS) [39] is a middleware for robot software development. It consists on a collection of tools, libraries and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms.

One of ROS' main strengths is its ease in integrating the work developed by different teams, allowing for code reuse. The ROS ecosystem now consists of tens of thousands of users worldwide, working in domains ranging from tabletop hobby projects to large industrial automation systems.

ROS systems consist of several small computer programs that connect to one another and continuously exchange *messages*. These messages travel directly from one program to another, meaning there is no central routing service. In addition to this, ROS chose a multilingual approach, meaning that it allows working several different languages, such as Python, C++, MATLAB, Java and others. ROS software modules can be written in any language for which a client library has been written. ROS conventions also encourage contributors to create standalone libraries and then wrap those libraries so they can send and receive messages to and from other ROS modules.

The core of ROS is released under the BSD license, which allows commercial and noncommercial use. Commercial systems often have several closed source modules communicating with a large number of open source modules, whereas academic systems are often fully open source.

2.3.2 ROSPlan

ROSPlan [8] is a framework which provides a collection of tools for AI planning in a ROS system. ROSPlan possesses a variety of nodes which encapsulate planning, problem generation and plan execution.

In Fig. 2.6, it is possible to visualise the nodes available in ROSPlan. The **Knowledge Base** is used to store a PDDL model. The **Problem Interface** is used to generate a PDDL problem and publish it on a topic or write it to a file. The **Planner Interface** is used to call a planner and publish the plan to a topic

[1]`http://wiki.ros.org/kinetic`

or write to a file. The **Parsing Interface** is used to convert a PDDL plan into ROS messages, ready to be executed. The **Plan Dispatch** encapsulates plan execution.

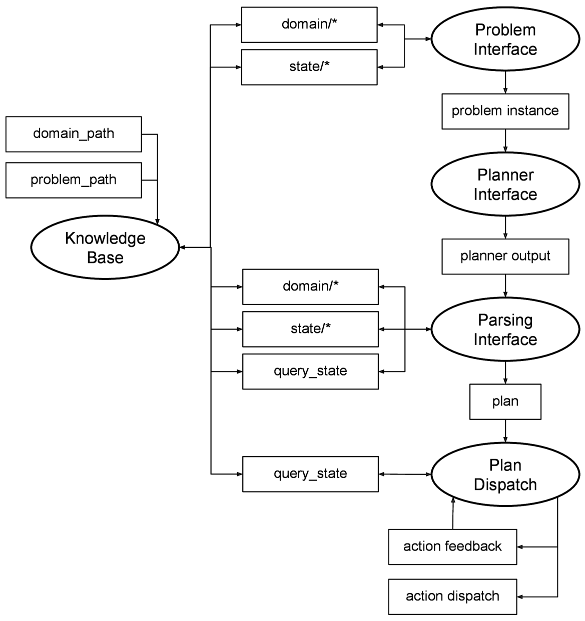


Figure 2.6: Overview of all the nodes in ROSPlan. The **Knowledge Base** stores the domain and the state. The **Problem Interface** uses the domain and the state to create an instance of the problem. The **Planner Interface** uses a planner to solve the problem instance. The **Parsing Interface** converts the PDDL plan into ROS messages that can be used by the Plan Dispatch. The **Plan Dispatch** executes the actions from the plan and receives feedback from the actions, which is then used to update the state of the world in the Knowledge Base [8].

This means that ROSPlan is able to parse PDDL domains and problems, as well as generate a PDDL plan and convert it to ROS messages. This is especially useful considering one can simply use this platform to handle all the interface between PDDL and ROS, allowing one to focus on the planner instead. In addition to this, ROSPlan uses POPF [10], a forward-chaining partial-order planner, as its default planner.

ROSPlan is open-source and its source code can be consulted on its *GitHub* repository [2].

[2] https://github.com/KCL-Planning/ROSPlan

Chapter 3

Approach and Methodology

This chapter focuses on the Olisipo algorithm and its implementation. First, section 3.1 starts by providing a high-level overview of Olisipo. The Olisipo algorithm is composed of an offline and on online phase. Section 3.2 details the offline phase of Olisipo, where an adaptable partially-ordered plan is generated from a totally-ordered plan (sub-section 3.2.1). Then, section 3.3 focuses on the online phase of Olisipo and on all of its separate steps. These steps can be split into two sub-phases: the Plan Generation Sub-Phase (PGen) and the Dynamic Plan Execution Sub-Phase (PExec). Starting by the PGen, sub-section 3.3.1 explains how the set of totally-ordered plans is built with a Branch & Bound search. Then, sub-section 3.3.2 focuses on what a Dynamic Bayesian Network is, on how a plan's success probability is calculated by resorting to it and on how it can be simplified in order to simplify the calculation. Sub-section 3.3.3 explains the algorithm used to iteratively build the Dynamic Bayesian Network. Sub-section 3.3.4 explains how a plan's success probability is calculated iteratively, as the Dynamic Bayesian Network is being constructed. Regarding the PExec, sub-section 3.3.5 goes over the process of checking whether a plan is compatible with the state of the world and on how the compatible lattermost action of the plan is selected.

3.1 High-level Overview of Olisipo

The Olisipo algorithm is an extended version of the algorithm originally developed by Oscar Lima, Michael Cashmore, Daniele Magazzeni, Andrea Micheli and Rodrigo Ventura [33]. This extended version was named *Olisipo* because it was developed in the University of Lisbon and *Olisipo* was the pre-Roman name for the city of Lisbon. The flow from planning to execution of Olisipo can be visualised in Figure 3.1. The approach is essentially composed of an offline and an online phase. In this thesis, the offline phase remains unchanged from the original paper [33], while the online phase and executor suffered major changes.

The offline phase, performed previously to the robot starting plan execution, is responsible for converting a totally-ordered plan $\prod^{tt}$, generated by an external planner, to an adaptable partially-ordered plan $\prod'$. The totally-ordered plan $\prod^{tt}$ was defined in sub-section 2.2.1 and the partially-ordered plan $\prod'$ is defined below, in sub-section 3.2.1. The adaptable partially-ordered plan is passed on to the online phase. The offline calculation of the partially-ordered plan simplifies the search for a totally-ordered plan at runtime.

The online phase has two sub-phases: the Plan Generation Sub-Phase (PGen) and the Dynamic Plan Execution Sub-Phase (PExec). The PGen is used to generate the totally-ordered plan Γ with the highest probability of a successful execution, taking into consideration the state of the world S_0 and the partially-ordered plan $\prod'$. In addition to this, the first action of Γ is dispatched for execution. The PExec concerns the dynamic and robust execution of Γ, allowing for the skipping and repetition of actions.

Let us now consider the flow of the online phase of Olisipo and on how its two sub-phases are chained. This explanation is better understood by following along the reasoning with Figure 3.1. First, the robot reads the state of the world, S_0 and checks whether a totally-ordered plan, Γ, has already been determined or not. At start, no Γ has been determined so the algorithm enters the PGen. In the PGen, the algorithm uses the *Total-Order Extractor* to build a set of totally-ordered plans. Then, the *Plan Selector* chooses the totally-ordered plan with the highest success probability and sets it as the totally-ordered plan to be used, Γ. Lastly, the first action of Γ is dispatched for execution. After executing this action, the robot reads again the state of the world and checks whether a totally-ordered plan has already been found. Since now Γ has already been found, the algorithm advances to the PExec. In this sub-phase, it first checks whether S_0 is compatible with Γ. If it is not compatible, then the PGen is repeated and a new Γ is computed. If it is compatible, then the *Compatible Lattermost Action Selector* selects the compatible lattermost action and dispatches it for execution. The robot will keep on reading the state of the world, and executing PGen and PExec, until the state of the world matches the goal.

3.2 Offline Phase

3.2.1 Generating an Adaptable Partially-Ordered Plan

After a totally-ordered plan $\prod^{tt}$ is generated by an external planner, it is converted into a partially-ordered plan $\prod$ by generating a node for each instantaneous action and each durative action start and end. For each node n_1 that supports the conditions of another node n_2, the following temporal relation is generated: $0 < [time(n_2) - time(n_1)] < \infty$. For each pair of nodes $\langle n_1, n_2 \rangle$ representing the start and end of a durative action a, a relation representing the constraints in $dur(a)$ is generated. For each pair of actions $\langle a_1, a_2 \rangle$ that interfere (if they have conflicting effects or if the effects of one action conflict with the conditions of another), an interference relation is generated. As durative actions are represented by two nodes (*at start* and *at end*), the relation is generated between the nodes that contain the interfering effects or conditions. In the case of an interfering *over all* condition, a relation is generated for both nodes.

This way, the plan $\prod$ is generated. Before starting plan execution, the causal edges of $\prod$ are removed to give more flexibility in action selection, therefore resulting in the partially-ordered plan $\prod'$.

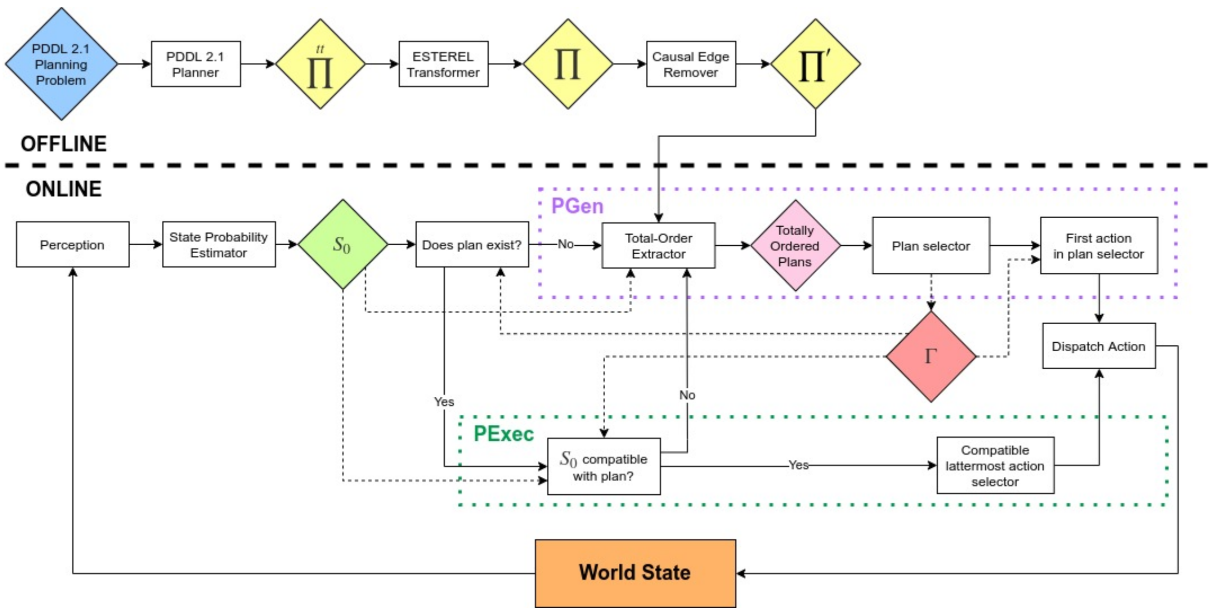


Figure 3.1: High-level overview of Olisipo's flow from planning to execution. The offline phase consists of converting a totally-ordered plan $\prod^{tt}$ into the partially-ordered plan $\prod'$. The online phase consists of two sub-phases, PGen and PExec. The PGen consists of finding the totally-ordered plan with the highest success probability, Γ, and dispatching its first action for execution. The PExec occurs when Γ has already been found and is compatible with the state of the world S_0. If Γ is compatible with S_0, then the compatible lattermost action of Γ is dispatched for execution.

Taking as an example the time-triggered plan $\prod_{ex}^{tt}$ of Figure 3.2, which was taken from [33], it is possible to generate the partially-ordered plan $\prod_{ex}$ of Figure 3.3 and, after removing the causal edges, to obtain the $\prod'_{ex}$ of Figure 3.4.

```
0.000:  (goto r0 wp1 m0)              [14.000]
0.000:  (goto r1 wp0 m0)              [ 9.000]
14.001: (switch_on r0 m0)             [ 5.000]
19.002: (load_at_machine r1 r0 m0)    [15.000]
34.002: (goto r1 m0 wp1)              [14.000]
48.002: (ask_unload r1 wp1)           [ 5.000]
53.003: (wait_unload r1 wp1)          [15.000]
```

Figure 3.2: Example of a time-triggered plan, generated by an external planner. This plan contains the actions to be dispatched at each time instant, with the first column being the dispatch time of each action and the third column being its duration. [33]

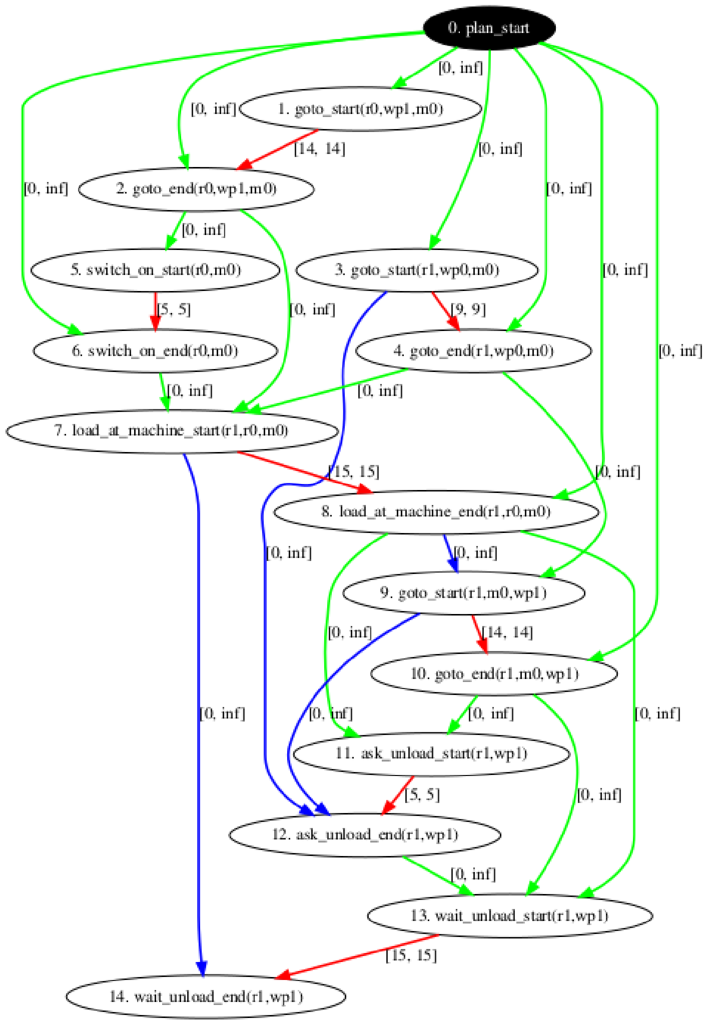


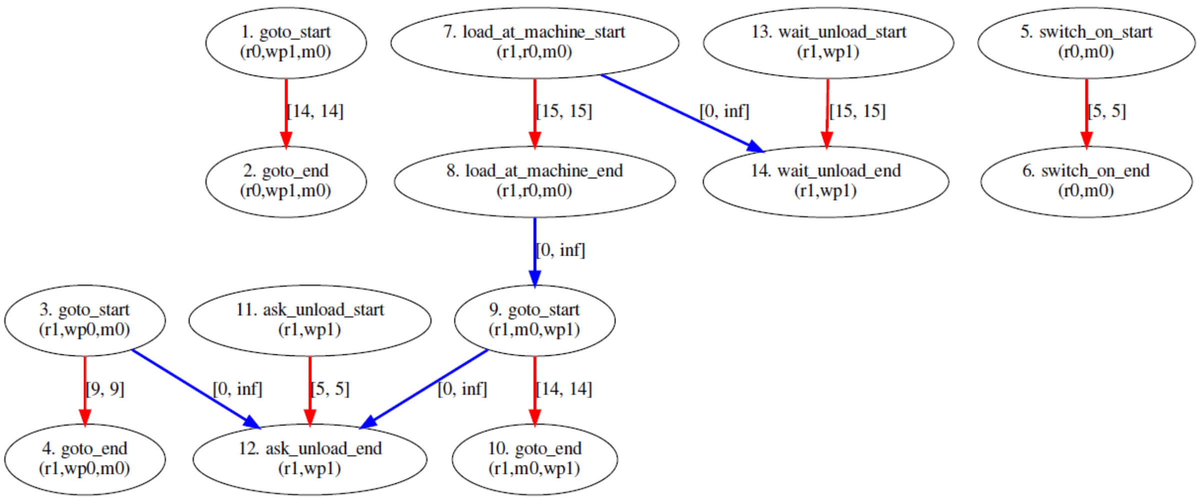
Figure 3.3: Graph of the partially-ordered plan $\prod_{ex}$, obtained from the time-triggered plan of Figure 3.2. Nodes represent instantaneous actions, or the start or end of durative actions. Green edges illustrate causal support, red edges illustrate action duration and blue edges illustrate interference constraints [33].

3.3 Online Phase

3.3.1 PGen: Building the Set of Totally-Ordered Plans

The set of totally-ordered plans is built using a Branch and Bound (B&B) search through the nodes of $\prod'$, excluding the action start nodes which have already been dispatched but their respective action end nodes have not. This enables the repetition of an already executed action but prevents the algorithm from dispatching the action start of an action that has already started but has not finished yet.

Essentially, the search takes every node in $\prod'$ and orders them sequentially, hence building a totally-



1. goto_start (r0,wp1,m0)
7. load_at_machine_start (r1,r0,m0)
13. wait_unload_start (r1,wp1)
5. switch_on_start (r0,m0)
[14, 14]
[15, 15]
[0, inf]
[15, 15]
[5, 5]
2. goto_end (r0,wp1,m0)
8. load_at_machine_end (r1,r0,m0)
14. wait_unload_end (r1,wp1)
6. switch_on_end (r0,m0)
[0, inf]
3. goto_start (r1,wp0,m0)
11. ask_unload_start (r1,wp1)
9. goto_start (r1,m0,wp1)
[9, 9]
[0, inf]
[5, 5]
[0, inf]
[14, 14]
4. goto_end (r1,wp0,m0)
12. ask_unload_end (r1,wp1)
10. goto_end (r1,m0,wp1)

Figure 3.4: Graph of the adaptable partially-ordered plan $\prod'$ after removing the causal edges from $\prod$ [33].

ordered plan. An example of this search can be seen in Figure 3.5. In this figure, the search begins from the initial state S_0 and explores in a depth-first manner. Action nodes 1 and $1''$ are applicable from S_0. Node 1 is first chosen to be applied and it yields state S_1. Then, node 2 is applied and the probability of actions 1 and 2 being both successful, when applied to state S_0, is $P(1, 2) = 0.70$. This probability is calculated with a Dynamic Bayesian Network formally defined below, in sub-section 3.3.2. From state S_2, action 3 can be applied, therefore leading to state S_3, which matches the goal. Now, we have found our first reference value to be used in the rest of the search. The value used as reference is the probability of actions 1, 2 and 3 being successful and of the facts of the goal being present in S_3, $P(1, 2, 3, G \subset S_3) = 0.40$. Now, while performing the rest of the search, a certain branch can be pruned if the joint probability of the actions is lower than the success probability of the best previously found solution, 0.40 in this case. Hence, the search is pruned at states S'_4 and S'_2.

To predict the state resulting from the execution of a given action, an **action simulator interface** is used. In the example of Figure 3.5, the action simulator interface identifies that actions 1 and $1''$ are applicable to S_0. In addition to this, it also predicts that S_1 is the predicted state resulting from the successful execution of action 1 to state S_0. In the same manner, S_2 is the predicted state resulting from the execution of action 2 to state S_1, which is also a predicted state by itself as mentioned before.

The theoretical foundation on how to calculate a plan's success probability using a Dynamic Bayesian Network is presented in sub-section 3.3.2. However, one concept we can already introduce, that will be further explained on sub-section 3.3.2, is the fact that $P(A)$ is monotonically decreasing as more actions are added to A. In addition to this, we have that $P(A) \geq P(A \cap G)$, hence $P(A)$ is an upper bound for $P(A \cap G)$ and it can be used to prune the search.

The conditions to prune the search at a certain state S_i are:

- If state S_i matches the goal;
- If the joint probability of the actions leading to S_i is lower than the probability of the best previously found solution;
- If the sequence of actions leading to S_i violates the temporal constraints;
- If there are no more applicable actions to S_i.

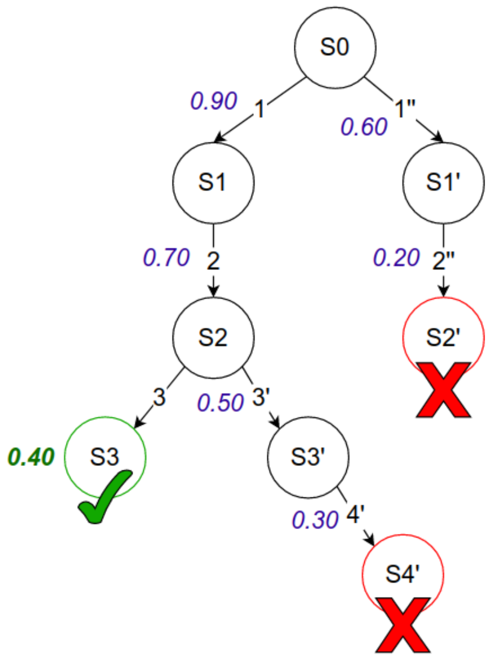


Figure 3.5: Example of the Branch and Bound search which builds the set of totally-ordered plans. Each node S_i represents a state of the world, with S_0 being the initial state of the world and the other nodes being predicted states of the world. The edges connecting the nodes represent actions, e.g. S_1 is the predicted state resulting from applying action 1 to S_0. The number to the side of each action represents the joint probability of all actions in the plan being successful, e.g. 0.70 is the joint probability of actions 1 and 2 being successful. The green mark below S_3 indicates that S_3 matches the goal state and a solution has been found. The probability of a solution corresponds to the joint probability of all previous actions being successful and of the facts from the goal being present in the solution, hence $P(1, 2, 3, G \subset S_3) = 0.40$. In the remaining search, when the joint probability of the actions becomes lower than the probability of the best solution found so far, the search is pruned. This pruning occurs in S'_4 and S'_2.

3.3.2 PGen: Calculating a Plan's Success Probability

A plan's probability of success is calculated resorting to a Dynamic Bayesian Network (DBN). The concept of a DBN was first introduced in [11] and it consists of a Bayesian Network that represents a temporal probabilistic model [40]. In a DBN, nodes are organised along time, meaning that the time structure is explicit in the network and that variables are related to each other over adjacent time steps. This DBN is built under a set of rules explained below, which allow us to take several conclusions from its structure.

Our intention is to compute the success probability of a totally-ordered plan. We assume that it is possible to determine whether the execution of an action was successful. In addition to this, we define a plan's success as the conjunction of the success of all actions and the achievement of the goal at the final state of the plan.

Let $F = [a_1, ..., a_N]$ be a totally-ordered plan where $a_i \in A_{start} \cup A_{end} \cup A_{inst}$, with A_{start}, A_{end} and A_{inst} being the sets of all possible ground start, ground end and ground instantaneous actions in the domain, respectively.

Let the projection operator δ map these ground actions to the set A, that is, both start and end actions in F map to the same action in A, while instantaneous actions are mapped one to one to A, $\delta : A_{start} \cup A_{end} \cup A_{inst} \rightarrow A$.

A proposition p takes a probabilistic value in the interval [0, 1], where 1 implies certainty in that proposition being *True* and 0 implies certainty in it being *False*. The execution of each action a in the plan also has a probability of success ϕ_a = [0, 1]. Upon a successful execution of a, each effect proposition $p \in$ *eff(a)* has a probability $\psi_a(p)$, where $\psi_a(p) = 1$ represents a positive effect (adding a proposition to the world) and $\psi_a(p) = 0$ a negative effect (removing a proposition from the world). Hence, it is important to point out that we separate the action success probability ϕ from the probability of the effects of that action $\psi_a(p)$.

Let us now look more specifically into the **Dynamic Bayesian Network** and its structure. Each node in the DBN defines a random variable defined by a CPD, conditioned by its parents. Nodes without parents correspond to unconditioned random variables. Let us denote nodes with capital letters and let the parents of node X be denoted by $Pa(X)$. The CPD of node X is given by the distribution $P(X|Pa(X))$.

In the DBN, each state S_k is represented by a set of nodes, each one representing the probability of a proposition being true at time step k. Each one of these sets can be seen as a layer of nodes, one for each time step.

Between two successive time steps, say $k-1$ and k, we consider a single action node, corresponding to action a_k. Hence, we have two types of nodes in the DBN: propositional nodes, which model the probability of a proposition $p \in P$ at a discrete time step $k = 0, ..., N$; and action nodes, which model the execution of actions from the totally-ordered plan.

Let $\mathcal{S}$ be the set of all proposition nodes, $\mathcal{A}$ be the set of all action nodes, P be the set of ground propositions, A be the set of ground actions and π be the projection operator, where π: $\mathcal{S} \cup \mathcal{A} \rightarrow P \cup A$. This means that, when we apply the projection operator to a node $S \in \mathcal{S}$, we get the proposition associated with it $p \in P$, $\pi(S) = p$. If we apply the projection operator to an action node $T \in \mathcal{A}$, we get $a \in A$, $\pi(T) = a$.

The edges of the DBN are defined according to the following principles:

- Each node $S \in \mathcal{S}_k$ propagates to the corresponding node $S' \in \mathcal{S}_{k+1}$, where $\pi(S) = \pi(S')$, modelling the probability of the proposition changing truth value value from state $\mathcal{S}_k$ to $\mathcal{S}_{k+1}$, hence there is an edge from S to S' for all $k = 0, ..., N-1$;
- Each node $T \in \mathcal{A}$ has as parents the nodes corresponding to its preconditions and, as children, the nodes corresponding to its effects;
- A durative action appears split into an action start and end nodes, T_i and T_j respectively, where $i < j$, and each node has its specific preconditions and effects, with T_i being a parent of T_j.

The parents of each node are defined as follows:

- The parents of a node $S \in \mathcal{S}_k$ are: $Pa(S) = \{S^- \in \mathcal{S}_{k-1} : \pi(S^-) = \pi(S))\} \cup \{T \in [\mathcal{S} : \pi(T) = a_k] \wedge [\pi(S) \in \textit{eff}(a_k)]\}$; where $\textit{eff}(a_k)$ corresponds to the *at start* or *at end* effects of a certain action, depending whether a_k is a *start* or *end* action;
- The parents of each node $T \in \mathcal{A}$ where $\pi(T) = a_k \in F$ are:
  - If a_k is a *start* of a durative action: $Pa(T) = \{S \in \mathcal{S}_{k-1} : \pi(S) \in pre_{at-strart}(\delta(a_k))\}$;
  - If a_k is an *end* of a durative action, where $a_j \in F$ is the corresponding start action $\delta(a_k) = \delta(a_j)$, then: $Pa(T) = \{T \in \mathcal{S} : \pi(T) = a_j\} \cup \{S \in S_k : \pi(S) \in pre_{over-all}(\delta(a_k))\} \cup \{S \in S_{k-1} : \pi(S) \in pre_{at-end}(\delta(a_k))\}$

From this rules, two propositions can be inferred about the structure of the DBN:

- **Proposition 1**: Propositional nodes have at most one action as parent.

  Proof: All nodes in S_0 have no parent. Given a $k > 0$, the nodes in S_k correspond to the world state after the execution of a single action, a_k, hence there are no two actions with effects in S_k.

- **Proposition 2:** The DBN defined above is an acyclic graph.

  Proof: There are only two types of edges: edges from and to actions; and edges between proposition nodes. Hence, it is possible to infer that all edges in the DBN will point in the direction from S_{k-1} to S_k. Since no edges point in the direction of S_k to S_{k-1} or between nodes in S_k, we can conclude that no cycles can be formed and that the DBN is an acyclic graph.

Let us now focus on the CPDs of the nodes in the DBN. We have 5 types of nodes: propositional nodes without parents, which belong to S_0; propositional nodes with another propositional node as parent; propositional nodes which have an action node and a propositional node as parents; action nodes which correspond to the start of an action; action nodes which correspond to the end of an action.

S^-	$P(S\|S^-)$
F	p_{ft}
T	$1 - p_{tf}$

Table 3.1: CPD of a propositional node S with another propositional node as parent S^-.

T	S^-	$P(S\|T, S^-)$
F	F	p_{ft}
F	T	$1 - p_{tf}$
T	F	$\psi_{a_k}(\pi(S))$
T	T	$\psi_{a_k}(\pi(S))$

Table 3.2: CPD of a propositional node S with another propositional node S^- and an action node T as parents.

- For the nodes belonging to S_0, since we assume conditional independence between these nodes, each node has a probability of $\rho(\pi(S))$.
- If $S \in \mathcal{S}_k$ is a propositional node with $S^- \in \mathcal{S}_{k-1}$ as parent, then its CPD is defined by Table 3.1, where p_{ft} is the probability of a false proposition turning spontaneously true and p_{tf} is the probability of a true proposition turning spontaneously false.
- If $S \in \mathcal{S}_k$ is a propositional node with $T \in \mathcal{A}$ and $S^- \in \mathcal{S}_{k-1}$ as parents, where $\pi(S) = \pi(S^-)$, then its CPD is defined by Table 3.2.
- If T is an action node corresponding to the start of action a_k, then its CPD is defined by:

$$P(T|Pa(T)) = \phi_{a_k} \cdot I(pre(\pi(T))) \tag{3.1}$$

  where $I(\cdot)$ is the indicator function, which is 1 if all of the arguments are verified and 0 otherwise. ϕ_{a_k} is the success probability of action a_k.
- If T is an action node corresponding to the end of an action, then its CPD is defined by:

$$P(T|Pa(T)) = I(Pa(T)) \tag{3.2}$$

  where the parents of T include not only its precondition nodes, but also the corresponding action start node.

An example of the structure of the constructed DBN can be seen in Figure 3.6.

Now, with the DBN fully specified, we can compute the success probability of the totally-ordered plan F. We wish to calculate two different probabilities: the joint probability of all actions of F being

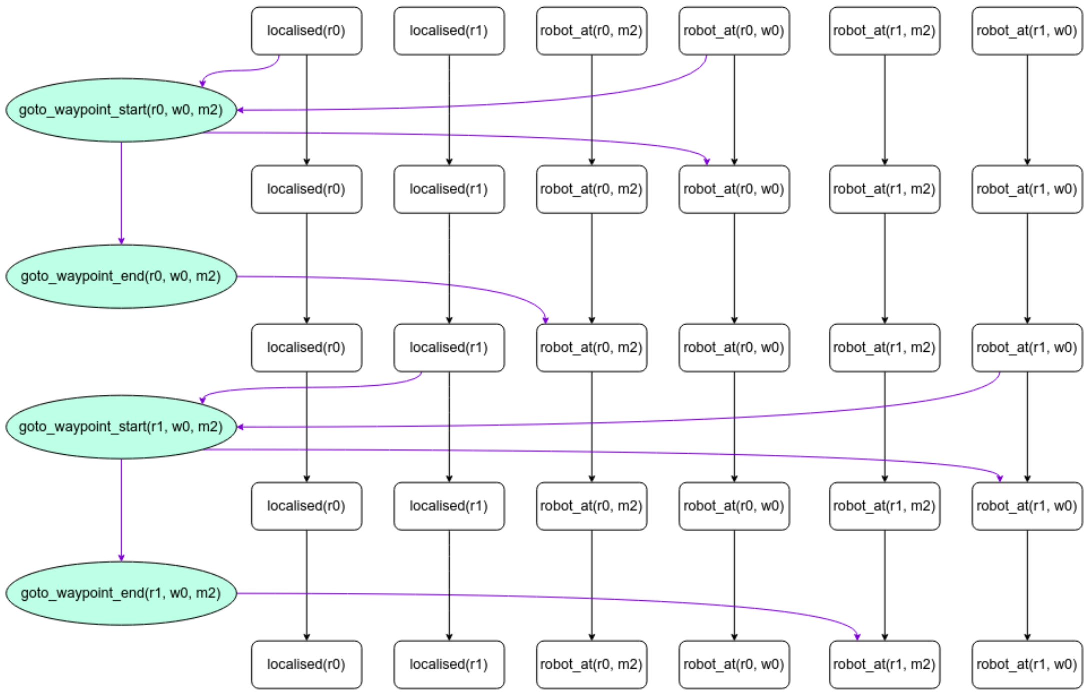

localised(r0)
localised(r1)
robot_at(r0, m2)
robot_at(r0, w0)
robot_at(r1, m2)
robot_at(r1, w0)
goto_waypoint_start(r0, w0, m2)
goto_waypoint_end(r0, w0, m2)
goto_waypoint_start(r1, w0, m2)
goto_waypoint_end(r1, w0, m2)

Figure 3.6: Example of the DBN's structure. Rectangular nodes symbolise proposition nodes and ellipsoidal nodes represent action nodes.

successful, $P(\mathcal{A})$; the joint probability of all actions of F being successful and of the goal propositions being in the final state, $P(\mathcal{A} \cap \mathcal{G})$ with $\mathcal{G} \subset \mathcal{S}_N$ being the set of nodes corresponding to the goal propositions in the last state. $P(\mathcal{A})$ can be obtained from Eq. 3.3, by the full joint distribution of the whole network, marginalised over all nodes that are not actions. $P(\mathcal{A} \cap \mathcal{G})$ can be obtained from Eq. 3.4, by the full joint distribution of the whole network, marginalised over all nodes that are neither actions nor goal propositions.

$$P(\mathcal{A}) = \sum_{\mathcal{S}_0,\ldots,\mathcal{S}_{N-1}} \sum_{\mathcal{S}_S} \prod_{S\in\mathcal{S}_0} \rho(\pi(S)) \prod_{T\in\mathcal{A}} P(T|Pa(T)) \prod_{S\in\mathcal{S}_1\cup\ldots\cup\mathcal{S}_N} P(S|Pa(S)) \tag{3.3}$$

$$P(\mathcal{A}\cap\mathcal{G}) = \sum_{\mathcal{S}_0,\ldots,\mathcal{S}_{N-1}} \sum_{\mathcal{S}_S\setminus\mathcal{G}} \prod_{S\in\mathcal{S}_0} \rho(\pi(S)) \prod_{T\in\mathcal{A}} P(T|Pa(T)) \prod_{S\in\mathcal{S}_1\cup\ldots\cup\mathcal{S}_N} P(S|Pa(S)) \tag{3.4}$$

From Equations 3.3 and 3.4, it is possible to infer two propositions:

- **Proposition 3**: $P(\mathcal{A})$ is monotonically decreasing.

  Proof: As more actions are considered in the calculation, more factors will be considered in the marginalisation of the actions, hence the computed value will reduce or stay the same. As an example:

$\prod_{T \in \mathcal{A}_2} P(T|Pa(T)) = P(a_1|Pa(a_1)) \cdot P(a_2|Pa(a_2)) \leq \prod_{T \in \mathcal{A}_1} P(T|Pa(T)) = P(a_1|Pa(a_1)$

- **Proposition 4**: $P(\mathcal{A}) \geq P(\mathcal{A} \cap \mathcal{G})$.

  Proof: This proposition comes from the fact that Equation 3.3 marginalises over a greater number of nodes than Equation 3.4. This can can be seen on the second sum of each Equation, since the sum $\sum_{S_S}$, from Equation 3.3, yield a greater or equal value than the sum $\sum_{S_S \setminus \mathcal{G}}$, from Equation 3.4.

This DBN can be simplified with the following 3 rules, without altering the value resulting from the probability calculation:

1. A node that is not the goal and is not a parent of any other node can be discarded by marginalisation.

   Proof: We marginalise over all nodes which are not the goal or the actions. If a node has no children, then no other node depends on its value and it does not affect the probability calculation. The example below shows the effect of marginalising over node a, which has no children.

$$\sum_{a,b,c,...} P(a|Pa(a)) \cdot \lambda(b,c,...) = \sum_{b,c,...} \lambda(b,c,...) \cdot \sum_{a} P(a|Pa(a)) = \sum_{b,c,...} \lambda(b,c,...) \tag{3.5}$$

2. The marginalisation sum corresponding to a node that is precondition of an action can be removed.

   Proof: When we calculate $P(\mathcal{A})$ or $P(\mathcal{A} \cap \mathcal{G})$, we are considering the probability of all actions being successful, in both cases. Since an action is only successful if all its preconditions are met, as defined by its CPD, we can only consider the case where each action parent matches that action's preconditions and discard that node's marginalisation.

3. For all proposition nodes with an action node T as parent, the edges from any other parent can be removed and the CPD replaced by $P(S|T) = \psi_{a_k}(\pi(S))$, where $\pi(T) = a_k$. The application of Rule 1 after Rule 3 can lead to further pruning.

   Proof: Let us consider a proposition node S who has as parents the proposition node S^- and the action node T, with $\pi(T) = a_k$. If $T = \textit{True}$ then the value of S only depends on the probability of the effects ψ_{a_k}, according to the CPD of Table 3.2. Since we are only interested in calculating the case where all actions are successful, the CPD of S can be simplified to $P(S|T) = \psi_{a_k}(\pi(S))$.

One important property derived from rule 3 is that each proposition node only has one parent, which can be another proposition node or an action node.

Figure 3.7 illustrates which nodes can be removed, according to the 3 pruning rules, from the DBN of Figure 3.6. Figure 3.8 presents the DBN without the removed nodes.

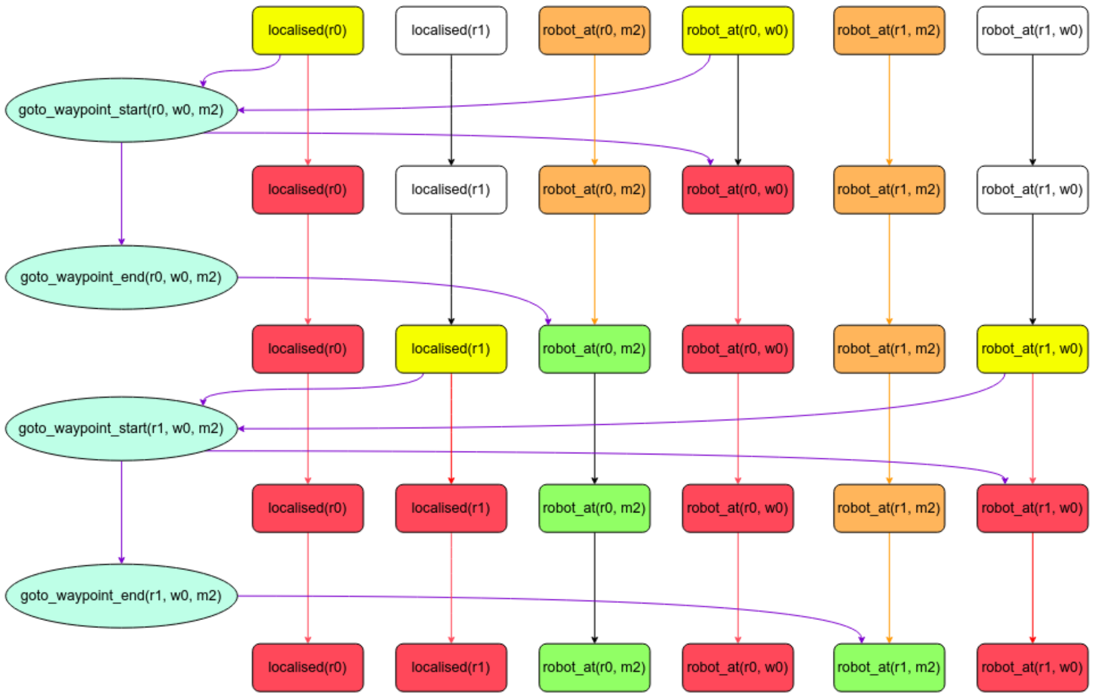

localised(r0)
localised(r1)
robot_at(r0, m2)
robot_at(r0, w0)
robot_at(r1, m2)
robot_at(r1, w0)
goto_waypoint_start(r0, w0, m2)
goto_waypoint_end(r0, w0, m2)
goto_waypoint_start(r1, w0, m2)
goto_waypoint_end(r1, w0, m2)

Figure 3.7: DBN representing which nodes can be removed from the DBN of Figure 3.6, by applying the 3 pruning rules. Rectangular nodes symbolise proposition nodes and ellipsoidal nodes represent action nodes. The colours of the nodes illustrate under which pruning rule they can be removed. Nodes and edges in red can be removed by Rule 1. Nodes in yellow have their marginalisation discarded, according to rule 2. Nodes and edges in orange can be removed after applying Rule 3, followed by Rule 1.

3.3.3 PGen: Building the Dynamic Bayesian Network Iteratively

Since we are building a DBN and calculating a probability at every step of the B&B search, the DBN is expected to be a bottleneck in terms of computing time and resources. Hence, it is vital to make this process efficient and iterative. Given this, instead of first building the entire network and then applying the pruning, we build the DBN already taking into account the pruning rules. Plus, the entire process is iterative, meaning that if we want to calculate the joint probability of actions a_1 and a_2 being successful after already having built the DBN and calculating $P(a_1)$, we need only to build the part of the DBN corresponding to a_2 and we can use the already calculated value $P(a_1)$ to simplify the calculation of $P(a_1, a_2)$.

Let us first focus solely on building the DBN iteratively.

We wish to directly build the DBN which would result from the application of the 3 pruning rules. This DBN, only contains nodes which were not removed by the 3 pruning rules. Hence, we wish to only add, to the DBN, nodes which are not removed by the pruning. With this in mind, let us again consider the plan $F = [goto\_waypoint\_start(r0, wp0, m2), goto\_waypoint\_end(r0, wp0, m2),$

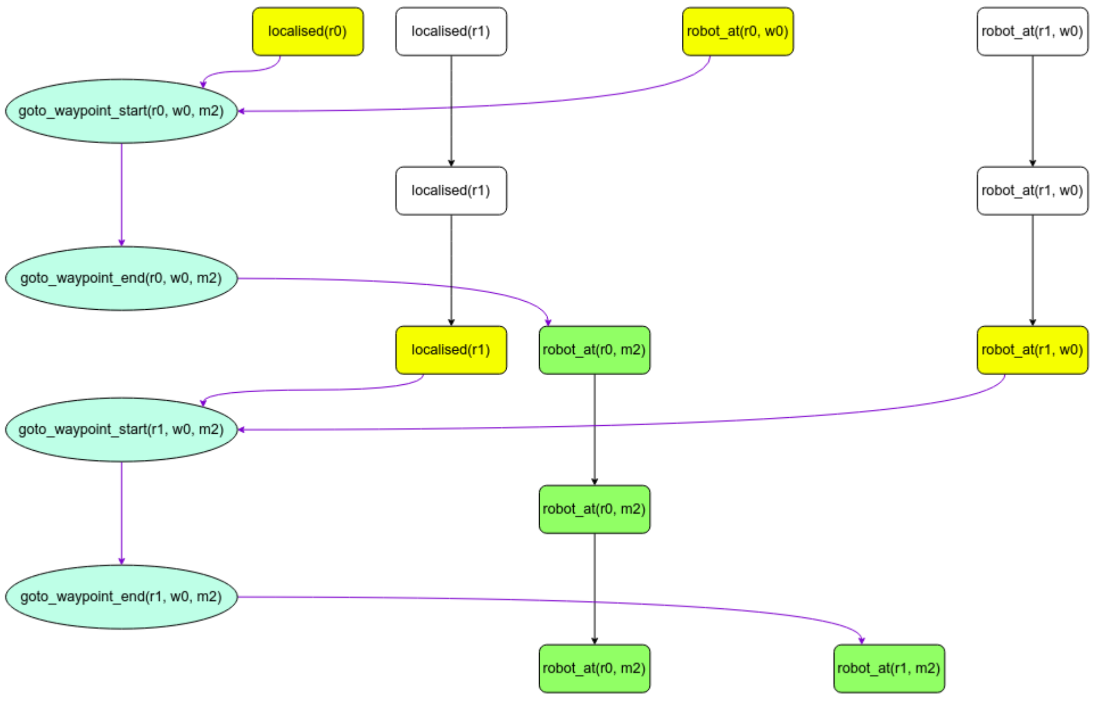
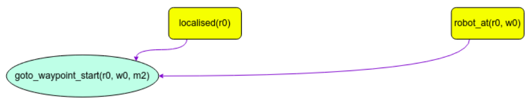


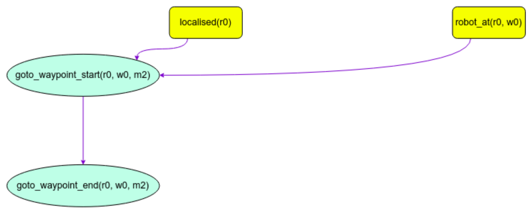
Figure 3.8: DBN after applying the 3 pruning rules to Figure 3.6. Rectangular nodes symbolise proposition nodes and ellipsoid nodes represent action nodes.

$goto\_waypoint\_start(r1, wp0, m2)$, $goto\_waypoint\_end(r1, wp0, m2)]$, which yields the DBN from Figure 3.8, after pruning. As the algorithm is performing the B&B search, it will add each action, one by one, to the DBN. When an action node, corresponding to action a_k, is added to the DBN, we add all nodes corresponding to its conditions to layer $k - 1$ and propagate those nodes to the previous layers, until reaching S_0 or a layer where it is an effect of the action preceding it. This process can be visualised in Figure 3.9, where each sub-figure illustrates the process of adding one action node to the DBN and the proposition nodes associated with it.

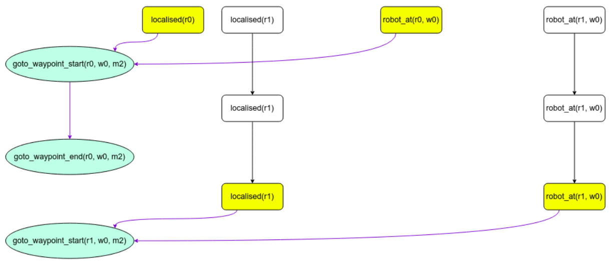
While the B&B search is performed and the nodes are added to DBN, the action simulator interface, mentioned in sub-section 3.3.1, is used to predict the state resulting from the successful execution of a certain action. When this interface detects that the predicted state matches the goal, then it stops the search and it adds the goal nodes to the last layer of the DBN (S_4 in the example of Figure 3.9d) and propagates them to the previous layers in the same manner as the condition nodes of an action, until reaching a layer where they are an effect of the action prior to that layer or until reaching S_0. This yields the DBN of Figure 3.8.



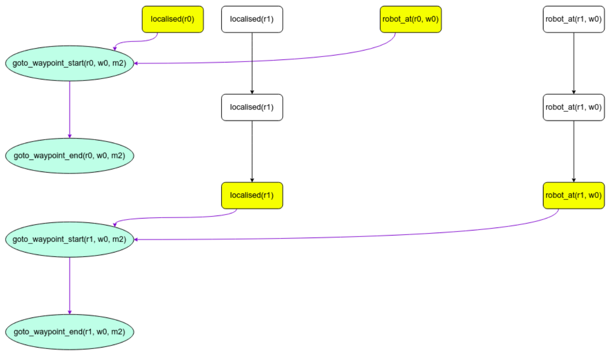
(a) DBN after adding node $goto\_waypoint\_start(r0, wp0, m2)$. The precondition nodes were added to S_0.


localised(r0)
robot_at(r0, w0)
goto_waypoint_start(r0, w0, m2)
goto_waypoint_end(r0, w0, m2)

(b) DBN after adding node $goto\_waypoint\_end(r0, wp0, m2)$, which has no parents other than the action start node.



(c) DBN after adding node $goto\_waypoint\_start(r1, wp0, m2)$. The precondition nodes were added to S_2 and propagated backwards until S_0.


localised(r0)
localised(r1)
robot_at(r0, w0)
robot_at(r1, w0)
goto_waypoint_start(r0, w0, m2)
localised(r1)
robot_at(r1, w0)
goto_waypoint_end(r0, w0, m2)
localised(r1)
robot_at(r1, w0)
goto_waypoint_start(r1, w0, m2)
goto_waypoint_end(r1, w0, m2)

(d) DBN after adding node $goto\_waypoint\_end(r1, wp0, m2)$, which has no parents other than the action start node.

Figure 3.9: Process of building iteratively the DBN. When an action node is added to the DBN, its parent nodes are also added (if they are not already present) and are propagated backwards, until reaching a layer S_k where they are a child of the action node a_k or until reaching S_0.

3.3.4 PGen: Calculating a Plan's Success Probability Iteratively

Since we are building the DBN iteratively, it is also necessary to calculate the desired probability iteratively, to avoid repeated calculations. Hence, after adding an action node and the corresponding proposition nodes to the DBN, Algorithm 1 can be used to calculate $P(\mathcal{A})$ for the action nodes present in the DBN. When the DBN is fully built, Algorithm 2 can be used to calculate $P(\mathcal{A} \cap \mathcal{G})$

Both Algorithms 1 and 2 make use of the global variable $joint\_prob$, which is initiated with the value 1 and then possesses the value of the joint probability of all actions already in the DBN. This is useful because part of the calculation for $P(a_1, a_2, a_3)$ is the same as $P(a_1, a_2)$, so we save the previous joint probability values and only need to calculate the value referring to the most recently added action node. To achieve this, we make not only use of $joint\_prob$ but also of $accounted\_nodes$, which is a list of all nodes already accounted in the calculation. $list\_probabilities$ is used to save all the values that $joint\_prob$ has taken, in order to allow backtracking in the network and only reusing the part of the DBN which remained after backtracking.

Essentially, the function $calculateActionsJointProbability$, from Algorithm 1, starts by getting the action's success probability and then multiplies it by the probability of each of its positive parents being $True$ and each of its negative parents being $False$. Lastly, it multiplies that value by $joint\_prob$ and sets the result of that product as the new $joint\_prob$. Plus, it adds $joint\_prob$ to $list\_probabilities$ and the action node to $accounted\_nodes$.

The function $calculateFullJointProbability$, from Algorithm 2, uses the fact that $calculateActionsJointProbability$ has already been called several times before and that $joint\_prob$ possesses the value $P(\mathcal{A})$, so it only needs to calculate the part that concerns the goals to determine $P(\mathcal{A} \cap \mathcal{G})$. Hence, it calculates the probability of each goal node being true in the last layer.

Algorithms 1 and 2 both make use of function $calculateColumn$, which can be found in Algorithm 3. This function calculates the probability of the proposition node $bottom\_node$ being $True$ or $False$, depending on the boolean value of the variable $true\_value$. Let us now look at how the function works.

First, $calculateColumn$ uses function $getTopPropositionParent$ to get the top parent of $bottom\_node$. The function $getTopPropositionParent$ checks the parent of $bottom\_node$, the parent of the parent of $bottom\_node$ and so on, until finding a proposition node which has no parents or has an action node as parent or is already in $accounted\_nodes$. When it finds such a node, it returns it. Taking Figure 3.8 as an example, the top parent of $localised(r1)$ from S_2 is $localised(r1)$ from S_0. The top parent of $robot\_at(r0, m2)$ from the last layer, S_4, is $robot\_at(r0, m2)$ from layer S_2. The top parent of $robot\_at(r1, m2)$ from the last layer is that node itself.

Then, if top_parent is already in $accounted\_nodes$, it sets $prob = 1$ because its probability is already included in $joint\_prob$ (line 5 and 6). If it is not, then it adds top_parent to $accounted\_nodes$ and checks if top_parent has a parent. If it has a parent and it is not in $accounted\_nodes$, then, according to the definition of function $getTopPropositionParent$, its parent can only be an action. So, it gets the probability of that node being $True$ knowing that its parent (the action node) is $True$ (line 10). If top_parent does not have a parent, then it can only be a node without parents (belonging to S_0) and we set $prob = \rho(\pi(top\_parent))$ (line 12).

Afterwards, we do the variable change $node = top\_parent$ and enter the while loop of line 16. This loop will transverse the chain of nodes between top_parent and $bottom\_node$. Inside the loop, function $getPropositionChildOf(node)$ checks the set of children of $node$ and returns the one which is a proposition node (line 17). Then we get the probability of that node becoming spontaneously false and spontaneously true (lines 18 and 19). Then, we can marginalise the probabilities of $node$ as done in line 25. However, if the parent of $node$ is a positive child of an action then we should only account for the probability of that parent being $True$ and remaining $True$ (line 21). If the parent of a node is a negative child of an action then we only account for the probability of that parent being $False$ and becoming $True$ (line 23). We are only accounting for a $True$ node remaining $True$ and a $False$ node becoming $True$ because the probability of either becoming $False$ is simply one minus the probability we obtained. This means that, as we go down the chain, we are always calculating the probability of each node being $True$. Lastly, we add $node$ to $accounted\_nodes$.

After exiting the $while$ loop, we add the $bottom\_node$ to $accounted\_nodes$ (line 34) and we return $prob$ if we want to know the probability of $bottom\_node$ being $True$ (lines 29, 30 and 35) or $1 - prob$ if we want the probability of $bottom\_node$ being $False$ (lines 31, 32 and 35).

The probabilities determined by this algorithm were verified by comparing them with the probabilities determined from the variable elimination algorithm [1] of [40].

Algorithm 1 Calculate Actions Joint Probability

```
1: global joint_prob                        ▷ Accumulated joint probability of all actions.
2: global list_probabilities                ▷ List of joint probabilities, for backtracking.
3: global accounted_nodes                   ▷ List of nodes already accounted in the calculation.

4: function calculateActionsJointProbability(action)
5:     prob = $\phi_{action}$
6:     for pos_parent of action do          ▷ Iterate over the positive parents of action.
7:         if pos_parent is proposition node then
8:             prob = prob*calculateColumn(pos_parent, True)
9:         end if
10:    end for
11:    for neg_parent of action do          ▷ Iterate over the negative parents of action.
12:        if neg_parent is proposition node then
13:            prob = prob*calculateColumn(neg_parent, False)
14:        end if
15:    end for
16:    joint_prob = joint_prob*prob
17:    list_probabilities.add(joint_prob)
18:    accounted_nodes.add(action)
19:    return joint_prob
20: end function
```

3.3.5 PExec: Checking if a Plan is Compatible with the State of the World & Dispatching the Compatible Lattermost Action

In the PExec sub-phase, we wish to check if the state of the world is compatible with the plan Γ. To achieve this, Algorithm 4 is used.

Algorithm 4 traverses the DBN of plan Γ from the last layer to the first one, from S_N to S_0. Then, it returns the first layer it finds that is compatible with the current state of the world. Hence, it will find the lattermost layer.

[1] https://github.com/aimacode/aima-python/blob/master/probability.py

Algorithm 2 Calculate Full Joint Probability

```
1: global joint_prob                    ▷ Accumulated joint probability of all actions.
2: global list_probabilities            ▷ List of joint probabilities, for backtracking.
3: global goal_                         ▷ List of facts in the goal.

4: function calculateFullJointProbability( )
5:     for goal_node in goal_ do
6:         prob = calculateColumn(goal_node, True)
7:         joint_prob = prob*joint_prob
8:     end for
9:     list_probabilities[-1] = joint_prob    ▷ Replace the last element of list_probabilities.
10:    return joint_prob
11: end function
```

The pruning rules remove all irrelevant proposition nodes from the DBN, so the nodes remaining in the DBN, at a certain layer, must be verified in the state of the world to enable the execution of the action node that follows that layer.

Let us now look in a more detailed way at how function *checkIfStateIsCompatibleWithPlan* from Algorithm 4 works. First, it checks whether the current state of the world, $\mathbb{S}$, entails the goal and has no actions executing (line 5). An action is executing if its action start node has already been dispatched for execution and its corresponding action end node has not. If these conditions are verified, then it simply returns the index of the last layer in the DBN. If it does not, then it starts traversing the layers of the DBN, from the layer before the last, S_{N-1}, to the first, S_0. If action node a_{i+1}, which follows layer i, corresponds to an action start and the propositions of S_i are verified in $\mathbb{S}$ (line 9 and 10), then action a_{i+1} is applicable and is dispatched for execution. Hence, we return i (line 11). If a_{i+1} is an end action node, then we need the state of the world to match the propositions at S_i and the corresponding action start needs to have already been executed (line 15). If both of these conditions are verified, we return i (line 16). If not, then the traversing through the layers continues. If the *for* loop finishes and no layer satisfied those conditions, then the function returns -1, meaning that the plan Γ is no longer compatible with the state of the world and a new Γ must be built, by running the PGen again.

As it can be seen in Figure 3.1, this verification will occur after the execution of each action. By considering the lattermost compatible action we are allowing for the skipping of actions. If the state of the world unexpectedly changes and it is no longer needed to execute a certain action, then the algorithm skips to the actions that need to be executed. One important detail is that, due to the formalism of *PDDL 2.1*, every action that has been started needs to be finished. So, even if the goal is already verified in the state of the world, the algorithm will first finish the actions that are still executing and only then declare that the goal has been reached. Another advantage that comes from using the pruned DBN is that we are only considering the propositions which are relevant for the execution of certain actions and for reaching the goal. If irrelevant propositions in the world change, then the same plan will remain valid. Plus, if the execution of a certain action fails, then the same plan might remain valid and the algorithm might retry the same action. Therefore, this algorithm can resist unexpected changes in the world and action failures without the need of replanning.

3.4 Implementation Takeaways

As a way to sum up what was presented on the Approach and Methodology chapter, we present a list with the main takeaways from Olisipo's implementation.

- Olisipo possesses an offline and on online phase;
- The offline phase consists in relaxing a totally-ordered plan into a partially-ordered plan. This partially-ordered plan is used to facilitate replanning at the online phase;
- The online phase consists of two sub-phases:
  - **Plan Generation Sub-Phase**: this phase consists in using the state of the world S_0 and the offline calculated partially-ordered plan $\prod'$ to build the totally-ordered plan with a highest success probability, Γ. A Branch and Bound search is used to accelerate the build of Γ. A plan's success probability is calculated by resorting to a Dynamic Bayesian Network. The Dynamic Bayesian Network is iteratively built. In the same manner, the probability of a totally-ordered plan is also iteratively calculated.
  - **Dynamic Plan Execution Sub-Phase**: this sub-phase consists in checking which is the lattermost action of Γ that is compatible with the state of the world S_0 and dispatching that action for execution.
- When Olisipo starts execution, the Plan Generation Sub-Phase will execute first, since no plan Γ has been found before. After executing this sub-phase, the Dynamic Plan Execution Sub-phase will execute Γ as long as it remain valid, considering the state of the world S_0. If Γ stops being valid, then the Plan Generation Sub-Phase is run again and a new Γ is determined. This cycle will repeat itself until the state of the world matches the goal.

Algorithm 3 Calculate Column

```
global accounted_nodes          ▷ List of nodes already accounted in the calculation
global cpds_map_                ▷ Dictionary that connects a nodes to its CPD

function calculateColumn(bottom_node, true_value)
    top_parent = getTopPropositionParent(bottom_node)
    if top_parent in accounted_nodes then
        prob = 1
    else
        accounted_nodes.add(top_parent)
        if top_parent has a parent then
            prob = cpds_map_[top_parent][$True$]
        else
            prob = $\rho(\pi(top\_parent))$
        end if
    end if
    node = top_parent
    while node $\neq$ bottom_node do
        node = getPropositionChildOf(node)
        spont_true_false = 1-cpds_map_[node][$True$]
        spont_false_true = cpds_map_[node][$False$]
        if nodeParentHasActionAsPositiveChild(node) then
            prob = prob*(1-spont_true_false)
        else if nodeParentHasActionAsNegativeChild(node) then
            prob = (1-prob)*spont_false_true
        else
            prob = prob*(1-spont_true_false)+(1-prob)*spont_false_true
        end if
        accounted_nodes.add(node)
    end while
    if true_value then
        returned_prob = prob
    else
        returned_prob = 1-prob
    end if
    accounted_nodes.add(bottom_node)
    return returned_prob
end function
```

Algorithm 4 Check if State is Compatible with Plan

```
global $\mathbb{S}$                                              ▷ Current state of the world.
global index_last_layer                                ▷ Index of the last layer on the DBN.
global actions_executing                                ▷ Set of actions under execution.

function checkIfStateIsCompatibleWithPlan( )
    if $G \subset \mathbb{S}$ and actions_executing $\neq$ [] then
        return index_last_layer
    end if
    for i = index_last_layer-1 to 0 do
        if $a_{i+1}$ is an action start node then          ▷ If action after layer $i$ is an action start node.
            if $S_i \subset \mathbb{S}$ then               ▷ If all proposition nodes in $S_i$ are verified in $\mathbb{S}$.
                return i
            end if
        else                                               ▷ If action after layer $i$ is an action end node.
            $a_{st}$ = getCorrespondingActionStart($a_{i+1}$)
            if $S_i \subset \mathbb{S}$ and $a_{st} \subset$ actions_executing then ▷ If all proposition nodes in $S_i$ are verified
  in $\mathbb{S}$ and the corresponding action start has already been executed.
                return i
            end if
        end if
    end for
    return $-1$
end function
```

Chapter 4

Experimental Results

This chapter focuses on the environment developed to test the Olisipo algorithm and on the results obtained.

4.1 The Simulation Environment

Since the main feature of Olisipo consists in coping with unexpected changes in the world and action failures, it was necessary to develop a simulation environment capable of replicating these events. Hence, the environment developed for the original paper [33], which would not account for state perturbations, was extended.

The extended environment uses ROSPlan [8] to parse the domain and problems files to the Knowledge Base. In addition to this, ROSPlan is also used to call the POPF [10] planner, which builds the totally-ordered plan $\prod^{tt}$ of Figure 3.1, and to dispatch the actions. ROS Services are used to handle the communication between different scripts. The ROS version utilised is ROS Kinetic Kame [1].

A Python script was developed to add and remove propositions from the Knowledge Base of the simulated environment. In addition to this, the previously developed C++ code was altered to account for action failures and for the probability of the effects of an action. These perturbations, performed by the Python and C++ scripts, take place after the execution of the start or end of an action. Plus, they occur in a random manner, with the user defining the probabilities of each proposition spontaneously becoming *False* and spontaneously becoming *True*, in a text file. The user also defines, in the same text file, the probability of success of each grounded action in the domain and the probability of each effect of each grounded action. This text file is read by the state perturbation Python script, by the C++ code and by the DBN, which uses it to define the nodes' CPDs.

[1]http://wiki.ros.org/kinetic

4.2 Esterel Dispatcher

To illustrate the advantages of the Olisipo algorithm, it is necessary to compare it to a frequently used alternative. A common way to deal with perturbed environments is to try to execute a totally-ordered plan and, when the execution fails, to replan and execute a new totally-ordered plan. Hence, this type of executor was developed. The Esterel Dispatcher, included in ROSPlan [8], was extended to account for state perturbations and action failures. The used planner was POPF [10], since it is also the same planner used in the offline phase of the Olisipo algorithm.

Putting it into clearer terms, the Esterel Dispatcher starts by reading the state of the world and using POPF to build a totally-ordered plan. Afterwards, it tries to execute this plan, ignoring changes in the state of the world, until the execution of an action fails, due to its conditions not being met or action failure. When the execution of an action fails, POPF is used to build a new totally-ordered plan, taking into account the new state of the world, and the Esterel Dispatcher tries to execute it. This cycle will repeat itself until the goal is achieved or until 10 replans are performed. Ten replans was defined as the maximum amount of allowed replans both for the Esterel dispatcher and for the Olisipo algorithm.

4.3 Factory Robot Domains

This environment consists in a robot responsible for the maintenance of several different machines on a factory, as displayed by Figure 4.1. At start, all machines are working and each machine has a different probability of breaking. If a machine breaks, then the production stops and the execution is declared a Failure. The robot's function is to perform maintenance on all machines and avoid the interruption of production. The robot has a probability ϕ_i of performing successful maintenance on machine i. Knowing that maintenance on machine i has been successfully performed, then machine i has a probability ψ_i of actually being maintained. Maintained machines can spontaneously stop being maintained with a probability p_{tf_i} and unmaintained machines can spontaneously be maintained, by a human maintenance team, with a probability p_{ft_i}. Plus, only machines which are working and are not maintained can break.

From this environment, two PDDL domains were developed to evaluate the performance of Olisipo and of the Esterel dispatcher. The choice was made to develop two separate domains to increase the heterogeneity in the results, meaning they would not be dependant on a specific domain with a specific number of actions and a specific number of propositions. The two developed domains were:

- **Simple Factory Robot Domain**, which is a simplified version possessing only one action *go_and_maintain_machine*;
- **Advanced Factory Robot Domain**, which possesses the actions *go_to_machine* and *maintain_machine*.

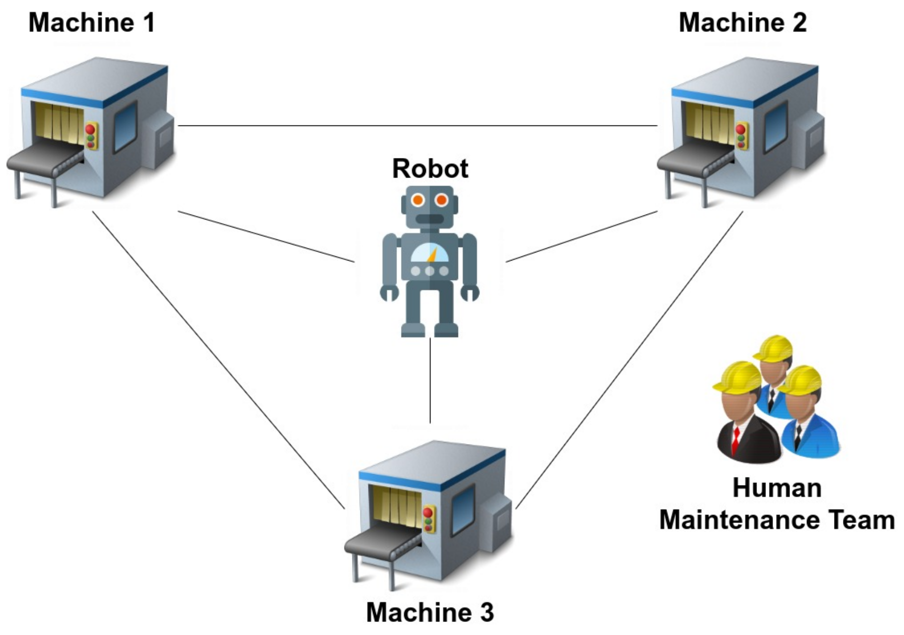


Figure 4.1: Illustration of the factory robot environment. In this environment, the robot is responsible for performing maintenance on all machines, to avoid the breakage of a machine. In addition to this, a human maintenance team might also unexpectedly maintain a machine.

With these domains, we expect the Olisipo algorithm to:

- Consider the probability of a machine breaking and to first maintain the machines most likely to break;
- Handle machines being unexpectedly maintained and skip the actions meant for it;
- Repeat actions if a machine unexpectedly stops being maintained;
- Not need to replan if an action fails and the current plan remains valid.

The chosen metrics to compare both dispatchers are:

- Rate of a successful execution;
- Average number of replans;
- Average number of actions executed.

A *bash* script was written to automate the process of running each dispatcher for several problems and for writing the results of its metrics to a CSV file. A Python script was developed to perform statistical analysis on the results.

4.4 Results

The Olisipo and Esterel Dispatchers were compared by performing 2000 trials on 10 different problems of *Simple Factory Robot Domain with 3 machines* (SF3) and 2000 trials on 8 different problems of *Advanced Factory Robot Domain with 3 machines* (AF3). Each problem was manually generated and represents a different perturbed environment. Table 4.1 and 4.2 present the probabilities of the different problems in SF3 and AF3, respectively.

	p1		p2		p3		p4		p5		p6		p7		p8		p9		p10	
Propositions	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}
machine_is_maintained_m1	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.05	0.02	0.05	0.02	0.03	0.04	0.02	0.04	0.04
machine_is_maintained_m2	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.05	0.02	0.05	0.04	0.02	0.02	0.02	0.02	0.02
machine_is_maintained_m3	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.02	0.05	0.02	0.05	0.04	0.03	0.20	0.00	0.05	0.06
machine_is_working_m1	0.00	0.02	0.00	0.02	0.00	0.02	0.00	0.02	0.00	0.02	0.02	0.03	0.02	0.04	0.00	0.02	0.00	0.02	0.00	0.05
machine_is_working_m2	0.00	0.06	0.00	0.06	0.00	0.07	0.00	0.08	0.00	0.09	0.02	0.06	0.02	0.08	0.00	0.07	0.00	0.07	0.00	0.07
machine_is_working_m3	0.00	0.08	0.00	0.10	0.00	0.12	0.00	0.14	0.00	0.16	0.02	0.09	0.02	0.12	0.00	0.04	0.00	0.09	0.00	0.11
Actions	ϕ	ψ	ϕ	ψ	ϕ	ψ	ϕ	ψ	ϕ	ψ	ϕ	ψ	ϕ	ψ	ϕ	ψ	ϕ	ψ	ϕ	ψ
go_maintain_machine_m1	0.75	0.85	0.75	0.85	0.75	0.85	0.75	0.85	0.75	0.85	0.70	0.80	0.70	0.80	0.60	0.75	0.60	0.60	0.65	0.75
go_maintain_machine_m2	0.75	0.85	0.75	0.85	0.75	0.85	0.75	0.85	0.75	0.85	0.70	0.80	0.70	0.80	0.70	0.70	0.60	0.75	0.65	0.75
go_maintain_machine_m3	0.75	0.85	0.75	0.85	0.75	0.85	0.75	0.85	0.75	0.85	0.70	0.80	0.70	0.80	0.60	0.75	0.70	0.75	0.55	0.65

Table 4.1: Probabilities definition of each problem (p1 to p10) from domain *Simple Factory Robot Domain with 3 machines*. p_{ft} represents the probability of a false proposition becoming spontaneously true and p_{tf} is the probability of a true proposition becoming spontaneously false. ϕ is the success probability of an action and ψ is the probability of the effects of that action.

From the results of the 2000 trials ran for each problem, Figures 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8 and 4.9 were built. Figures from 4.4 to 4.9 were built using Wolfram Mathematica.

To estimate the success probability, of each algorithm, the Wilson Score Interval [44] was used. Since each algorithm is either able or not able to solve a problem, its success distribution is a discrete binary distribution, namely a Bernoulli distribution. The Wilson Score Interval [44] can be used to estimate the probability p of a Bernoulli distribution, with one of its advantages being the fact that its cover probability is closer to the nominal value than the Normal Approximation Interval method. Hence, the probability of each algorithm solving a problem and its error is estimated by the Wilson Score Interval [44], presented in Equation 4.1.

In Equation 4.1, n is the total number of trials, n_s is the number of successful trials, n_f is the number of failed trials and z is the quantile function, with $z = 1.9599$ for a 95% confidence interval.

$$\hat{p} = \frac{n_s + \frac{1}{2}z^2}{n + z^2} \pm \frac{z}{n + z^2}\sqrt{\frac{n_s n_f}{n} + \frac{z^2}{4}} \tag{4.1}$$

Using Equation 4.1, it was possible to obtain the values presented in Tables 4.3 and 4.4. From these values, Figures 4.2 and 4.3 were built.

From Tables 4.3 and 4.4, it is possible to verify that, overall, the Olisipo algorithm achieved a higher

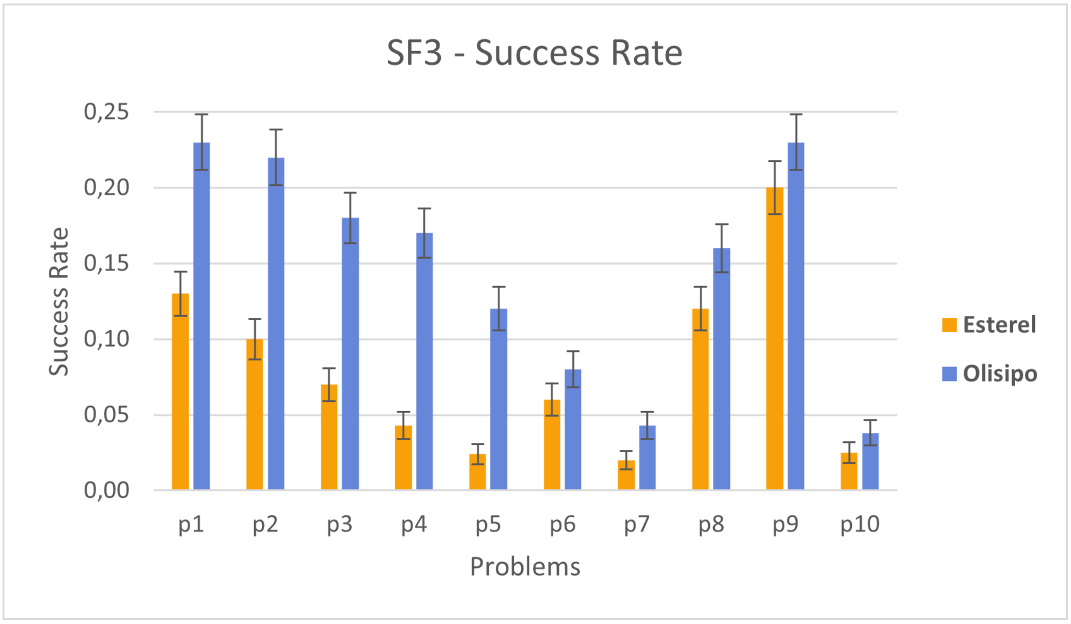


Figure 4.2: Bar chart of the estimated probability, as well as its error, of each algorithm solving each problem from the SF3 domain, calculated with the Wilson Score Interval from Equation 4.1.

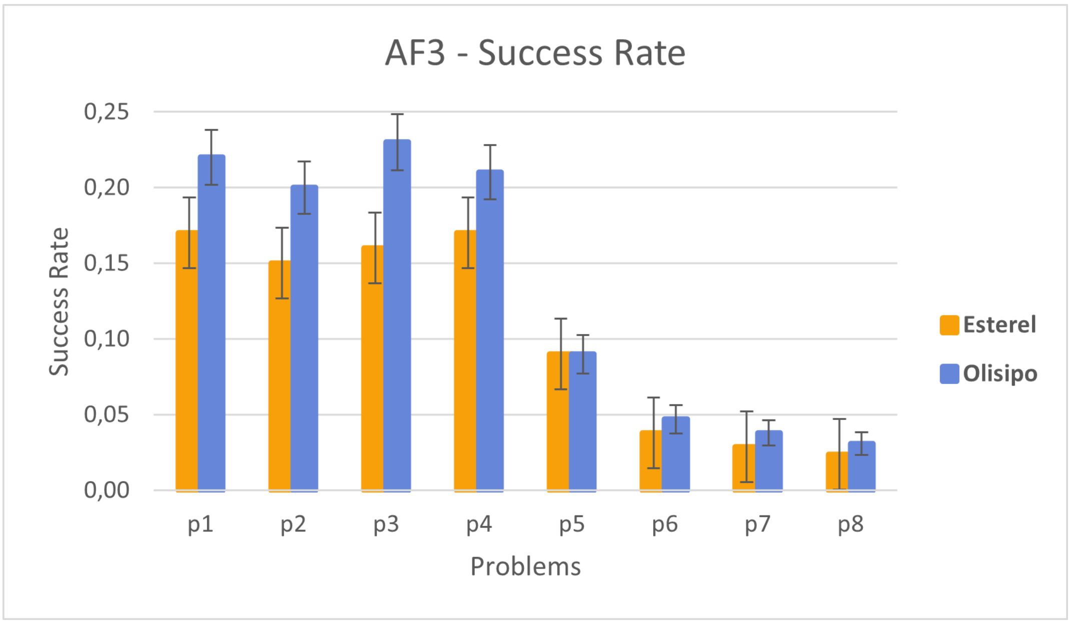


Figure 4.3: Bar chart of the estimated probability, as well as its error, of each algorithm solving each problem from the AF3 domain, calculated with the Wilson Score Interval from Equation 4.1.

	p1		**p2**		**p3**		**p4**		**p5**		**p6**		**p7**		**p8**	
Propositions	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}	p_{ft}	p_{tf}
machine_is_maintained_m1	0.02	0.05	0.02	0.10	0.02	0.15	0.02	0.20	0.02	0.05	0.02	0.06	0.02	0.07	0.02	0.08
machine_is_maintained_m2	0.05	0.02	0.10	0.07	0.15	0.12	0.20	0.17	0.05	0.02	0.06	0.02	0.07	0.02	0.08	0.02
machine_is_maintained_m3	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02
machine_is_working_m1	0.00	0.02	0.00	0.02	0.00	0.02	0.00	0.02	0.00	0.02	0.00	0.02	0.00	0.02	0.00	0.02
machine_is_working_m2	0.00	0.02	0.00	0.02	0.00	0.02	0.00	0.02	0.00	0.04	0.00	0.06	0.00	0.06	0.00	0.07
machine_is_working_m3	0.00	0.02	0.00	0.02	0.00	0.02	0.00	0.02	0.00	0.04	0.00	0.08	0.00	0.10	0.00	0.12
robot_at_m1	0.00	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
robot_at_m2	0.00	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
robot_at_m3	0.00	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Actions	ϕ	ψ	ϕ	ψ	ϕ	ψ	ϕ	ψ	ϕ	ψ	ϕ	ψ	ϕ	ψ	ϕ	ψ
maintain_machine_m1	0.70	0.75	0.75	0.85	0.75	0.85	0.75	0.85	0.75	0.85	0.75	0.85	0.70	0.80	0.70	0.80
maintain_machine_m2	0.70	0.75	0.75	0.85	0.75	0.85	0.75	0.85	0.75	0.85	0.75	0.85	0.70	0.80	0.70	0.80
maintain_machine_m3	0.70	0.75	0.75	0.85	0.75	0.85	0.75	0.85	0.75	0.85	0.75	0.85	0.70	0.80	0.70	0.80
go_to_machine_m1_m2	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75
go_to_machine_m1_m3	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75
go_to_machine_m2_m1	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75
go_to_machine_m2_m3	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75
go_to_machine_m3_m1	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75
go_to_machine_m3_m2	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75	0.70	0.75

Table 4.2: Probabilities definition of each problem (p1 to p8) from domain *Advanced Factory Robot Domain with 3 machines*. p_{ft} represents the probability of a false proposition becoming spontaneously true and p_{tf} is the probability of a true proposition becoming spontaneously false. ϕ is the success probability of an action and ψ is the probability of the effects of that action.

probability of success than the Esterel dispatcher. In the SF3 domain, the Olisipo algorithm presented an estimated improvement between -0.02 and 0.16, depending on the problem. In the AF3 domain, the Olisipo algorithm presented an improvement between -0.02 and 0.11. Despite the presence of some negative values, indicating a possible worst performance for the Olisipo algorithm, it is worth noting that on 12 of the 18 problems used, the improvement interval contained only positive values, indicating that Olisipo does present an improvement. Regarding the other problems, with the exception of problem 5 of AF3 which estimated the same probability and error for both dispatchers, at least 73% of the estimated interval is positive, indicating a higher probability of a successful execution for the Olisipo algorithm.

Figures 4.4 and 4.5 present the distribution of the number of replans, on successful executions, for each algorithm and for each problem. A successful execution occurs when a problem is successfully solved, with less or equal than 10 replans performed. In these Figures, it is possible to verify that, for successful executions, the mean of the number of replans of the Olisipo algorithm is consistently lower than the mean of the number of replans of the Esterel dispatcher, for both the SF3 and AF3 domains. The mean of the number of replans in successful executions for the Olisipo algorithm was consistently zero, whereas the Esterel dispatcher often needed to replan. The most accentuated differences were seen on problems 8 of both SF3 and AF3, where the mean of replans for the Esterel dispatcher was 2.2, while

SF3			
Problem	**Esterel**	**Olisipo**	**Improvement**
p1	0.13 ± 0.01	0.23 ± 0.02	$[0.07, 0.13]$
p2	0.10 ± 0.01	0.22 ± 0.02	$[0.09, 0.15]$
p3	0.07 ± 0.01	0.18 ± 0.02	$[0.08, 0.14]$
p4	0.043 ± 0.009	0.17 ± 0.02	$[0.10, 0.16]$
p5	0.024 ± 0.007	0.12 ± 0.01	$[0.08, 0.11]$
p6	0.06 ± 0.01	0.08 ± 0.01	$[0.00, 0.04]$
p7	0.020 ± 0.006	0.043 ± 0.009	$[0.008, 0, 038]$
p8	0.12 ± 0.01	0.16 ± 0.02	$[0.01, 0.07]$
p9	0.20 ± 0.02	0.23 ± 0.02	$[-0.01, 0.07]$
p10	0.025 ± 0.007	0.038 ± 0.008	$[-0.002, 0.028]$

Table 4.3: Table displaying the probabilities of the Esterel dispatcher and of the Olisipo algorithm solving each problem from the SF3 domain. These probabilities were calculated by making use of Equation 4.1. The last column displays the range of values where the estimated improvement, due to the use of Olisipo, lays.

AF3			
Problem	**Esterel**	**Olisipo**	**Improvement**
p1	0.17 ± 0.02	0.22 ± 0.02	$[0.01, 0.09]$
p2	0.15 ± 0.02	0.20 ± 0.02	$[0.01, 0.09]$
p3	0.16 ± 0.02	0.23 ± 0.02	$[0.03, 0.11]$
p4	0.17 ± 0.02	0.21 ± 0.02	$[0.00, 0.08]$
p5	0.09 ± 0.01	0.09 ± 0.01	$[-0.02, 0.02]$
p6	0.038 ± 0.008	0.047 ± 0.009	$[-0.008, 0.026]$
p7	0.029 ± 0.007	0.038 ± 0.008	$[-0.006, 0.024]$
p8	0.024 ± 0.007	0.031 ± 0.008	$[-0.008, 0.022]$

Table 4.4: Table displaying the probabilities of the Esterel dispatcher and of the Olisipo algorithm solving each problem from the AF3 domain. These probabilities were calculated by making use of Equation 4.1. The last column displays the range of values where the estimated improvement, due to the use of Olisipo, lays.

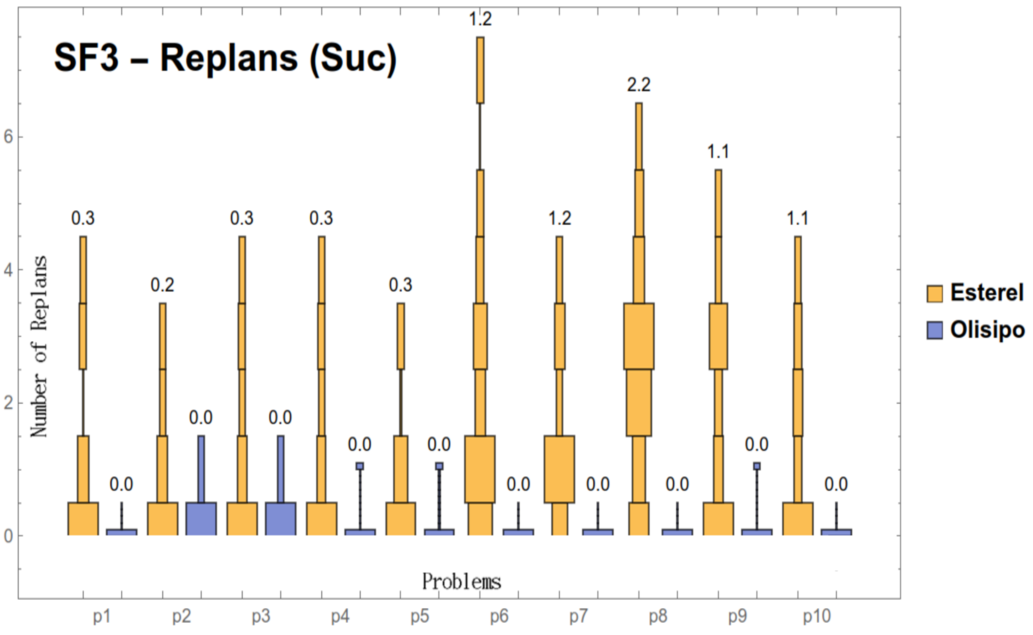


Figure 4.4: Distribution chart of the number of replans performed in the successful executions of the 2000 trials, performed for each problem of the SF3 domain, both for the Esterel and Olisipo algorithms. The number on top of each distribution corresponds to its median.

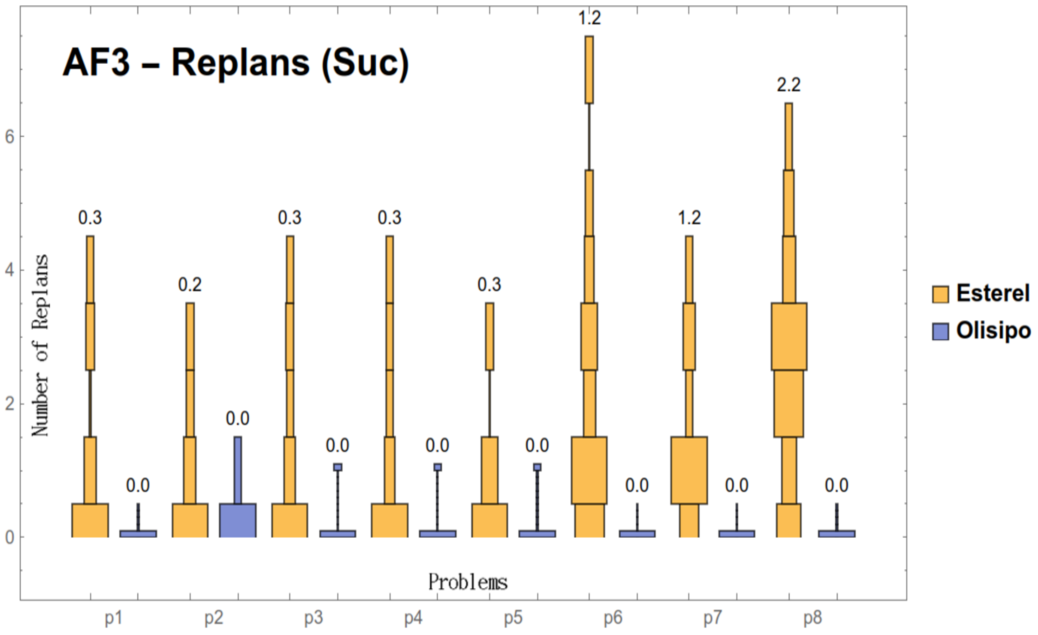


Figure 4.5: Distribution chart of the number of replans performed in the successful executions of the 2000 trials, performed for each problem of the AF3 domain, both for the Esterel and Olisipo algorithms. The number on top of each distribution corresponds to its median.

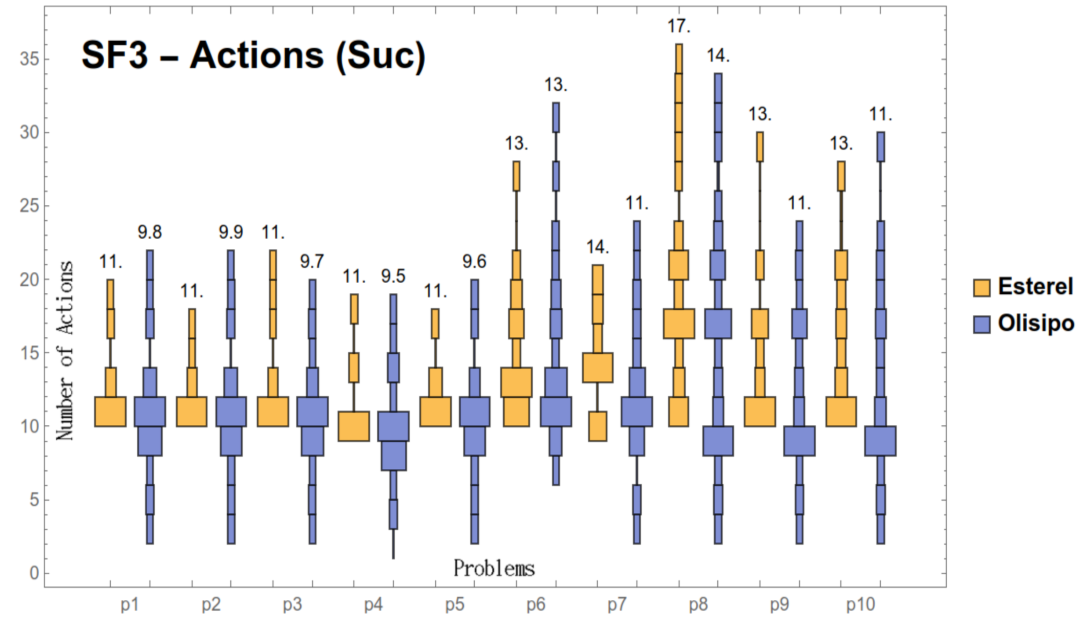


Figure 4.6: Distribution chart of the number of actions performed in the successful executions of the 2000 trials, performed for each problem of the SF3 domain, both for the Esterel and Olisipo algorithms. The number on top of each distribution corresponds to its median.

for the Olisipo algorithm it was 0. Another factor worth noting on Figures 4.4 and 4.5 is the shape of the distribution itself. For the Olisipo algorithm, it can be seen that almost all successful executions were achieved with zero replans. In contrast to this, the Esterel dispatcher has a much more spread out distribution, even achieving more than 5 replans on some trials of problems 6, 8 and 9 of SF3, as well as on some trials of problems 6 and 8 of AF3.

Considering the results from Figures 4.6 and 4.7, it is possible to verify that the mean for the number of actions in successful executions for the Olisipo algorithm was consistently lower than for the Esterel dispatcher. For all problems of SF3, except problem 6, the Olisipo algorithm offered a relative improvement of, at least, 10% in the mean of the number of actions for successful executions. The biggest relative improvements were achieved on problems 7 and 8, where the improvements were 21% and 18%, respectively. For all problems of AF3, the Olisipo algorithm offered an improvement of, at least, 28%. The biggest relative improvements were achieved on problems 6, 7 and 8, where the improvements were 48%, 47% and 52%, respectively. Regarding the shape of the distributions, it is worth noting that the Olisipo algorithm has a much more spread out distribution than the Esterel dispatcher, indicating that the expected number of actions in successful executions varies more on the Olisipo algorithm than on the Esterel dispatcher. However, the mode and mean for the Olisipo algorithm was always lower or equal than on the Esterel dispatcher.

Lastly, we will analyse the distribution of the number of actions on unsuccessful executions. These

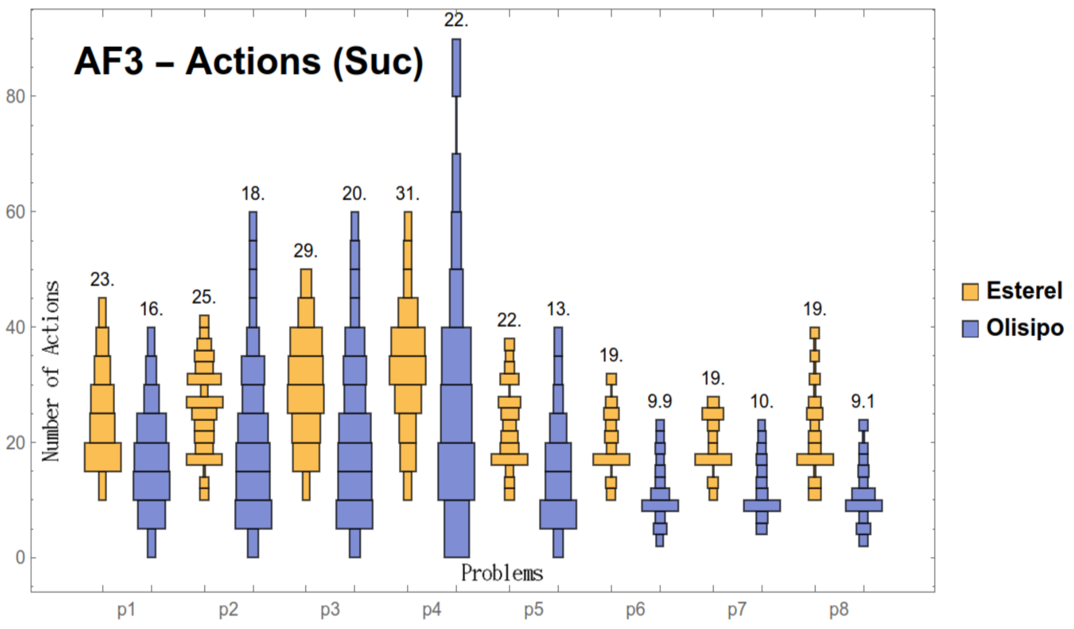


Figure 4.7: Distribution chart of the number of actions performed in the successful executions of the 2000 trials, performed for each problem of the AF3 domain, both for the Esterel and Olisipo algorithms. The number on top of each distribution corresponds to its median.

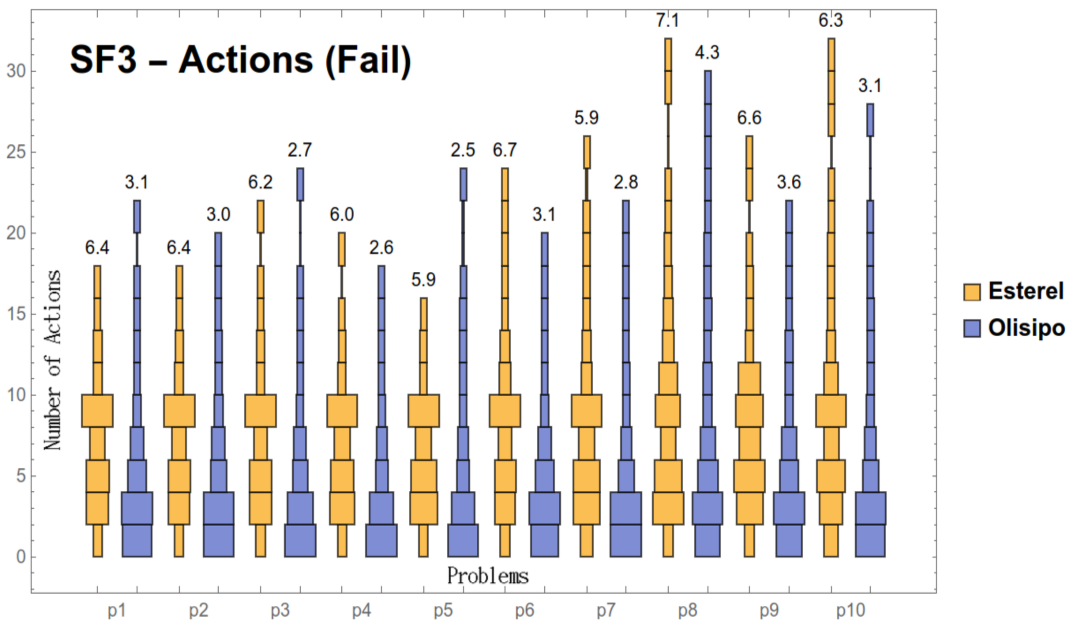


Figure 4.8: Distribution chart of the number of actions performed in the failed executions of the 2000 trials, performed for each problem of the SF3 domain, both for the Esterel and Olisipo algorithms. The number on top of each distribution corresponds to its median.

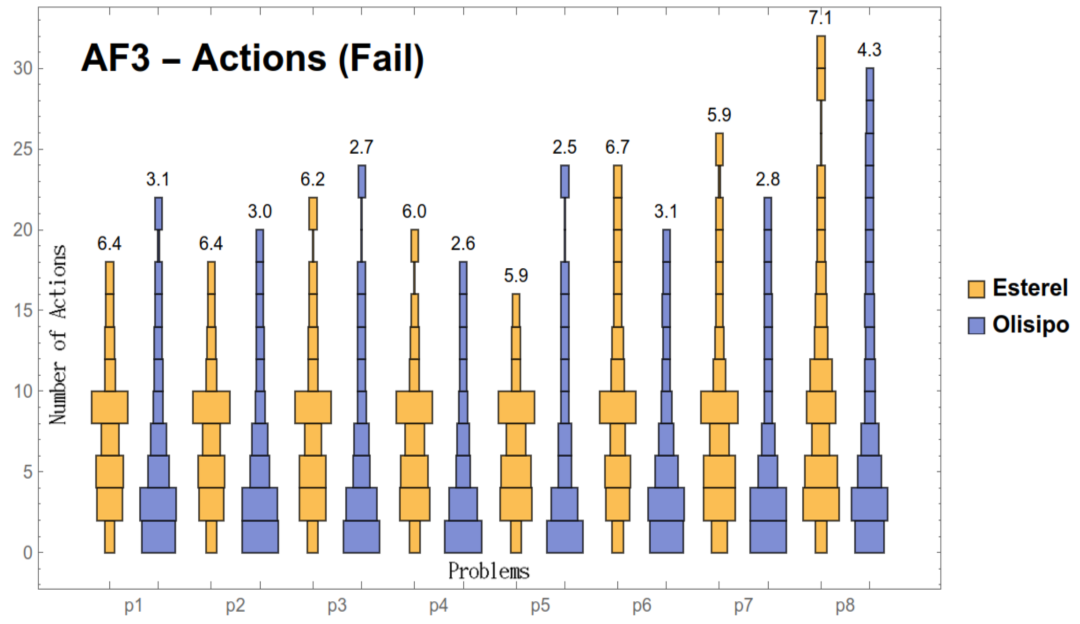


Figure 4.9: Distribution chart of the number of actions performed in the failed executions of the 2000 trials, performed for each problem of the AF3 domain, both for the Esterel and Olisipo algorithms. The number on top of each distribution corresponds to its median.

distributions can be visualised in Figures 4.8 and 4.9. The mean of executed actions before failing was lower on the Olisipo algorithm than on the Esterel dispatcher, for all problems of SF3 and AF3. The mode of the Olisipo algorithm was also lower or equal on all problems of SF3 and AF3. The mean of executed actions before failure on SF3 was improved by, at least, 39% with the Olisipo algorithm, in relation to the Esterel dispatcher. The biggest relative improvements occurred on problems 3, 4 and 5, where they were 56%, 57% and 58%, respectively. On the problems of AF3, the Olisipo algorithm also offered an improvement of at least 39% on the mean of executed actions before failure. The biggest improvements were achieved on problems 3, 4 and 5, with an improvement of 56%, 57% and 58% respectively.

4.5 Discussion

From these results, it is possible to conclude that, in comparison to a simple dispatcher, the Olisipo algorithm has a higher probability of solving a given problem, needs to replan less times to successfully solve a problem and executes less actions on successful and failed executions. The consistency in the obtained results is especially useful to support this claim, since the results were obtained from different problems and two different domains. This heterogeneity in the problems proves that Olisipo's performance improvement is not dependant on any problem's specificities. On the contrary, it remains for different problems and domains.

There were only two instances where the performance of Olisipo was similar to the Esterel dispatcher, which was the success probability of problem 5 of AF3 and the number of actions on successful executions of problem 6 of SF3. Regarding problem 5 of AF3, the mean of executed actions on successful executions was 13% less for Olisipo and the mean of executed actions on failed executions was 58% less. Regarding problem 6 of SF3, the Olisipo algorithm had a mean of 0 replans on successful executions whereas the Esterel had a mean of 1.2. Plus, the mean of executed actions on failed executions for the Olisipo algorithm was 54% less than for the Esterel dispatcher. Hence, even in problems where the performance of the Olisipo algorithm in one metric is similar to that of the Esterel dispatcher, the performance in the other metrics favour Olisipo.

Regarding the mean of executed actions in successful executions, the difference between Olisipo and Esterel was greater in AF3 than in SF3. Considering that SF3 only has one action on its domain, whereas AF3 has two actions, it is possible for this increase, in the improvement of Olisipo over Esterel, to be a trend. This trend would mean that Olisipo's improvement over a simple dispatcher, in this metric, would grow with the number of actions in the domain.

Regarding the number of replans on successful executions, the mean for Olisipo was consistenly zero for all problems. This means that Olisipo almost never needed to replan to achieve a successful execution. At the same time, this also means that Olisipo's replans rarely resulted in a successful execution. More tests to evaluate this metric would be interesting, since this could mean that repeating the PGen sub-phase does not achieve any additional successful executions.

To conclude this analysis, the Olisipo algorithm offered a consistent improvement over the Esterel dispatcher, on every metric and for every problem. Even in cases where the performance on one metric was similar for both Olisipo and Esterel, the performance on the other metrics favoured Olisipo. The improvement displayed by Olisipo in the number of executed actions in successful executions seems to increase with the number of actions in the domain. A domain with more actions could especially favour Olisipo. Regarding the improvement displayed in the number of replans on successful executions, Olisipo consistently did not need to replan, which indicates a considerable improvement over the Esterel dispatcher. However, this also indicates that the repetition of the PGen sub-phase rarely results in a successful execution.

Chapter 5

Conclusion

This thesis addresses the problem of bridging the gap between Classical Planning and Robotics. To achieve this, the algorithm developed for [33] was extended and further developed. The developed algorithm, named Olisipo, acts as an add-on to external planners, making their execution more robust in a real-world environment.

The Olisipo algorithm can be divided into an offline and an online phase. In the offline phase, the robot converts a totally-ordered plan, generated by an external planner, to a partially-ordered plan which can then be used during the online phase. In the online phase, the robot starts by reading the state of the world and building a totally-ordered plan, by making use of a B&B search with a DBN to iteratively calculate the success probability of a plan. Afterwards, the robot tries to execute this plan while it is compatible with the state of the world, ignoring irrelevant facts in the world and allowing for the skipping and repetition of actions. If the plan stops being compatible with the state of the world, then a new totally-ordered plan is generated and executed. Olisipo also supports a probabilistic description of the initial state of the world, when there is no certainty about the facts in the world.

To evaluate Olisipo, the simulation environment from [33] was extended to account for random state perturbations. The Esterel dispatcher, from ROSPlan [8], was also extended to account for state perturbations. The Esterel dispatcher consists in executing a totally-ordered plan until an action fails. Then, it builds a new totally-ordered plan and executes it until an action fails again. This simpler dispatcher was used as a benchmark to evaluate the performance of Olisipo.

The Factory Robot Environment was used to build the problems to be solved by both dispatchers. This environment consists in a robot responsible for the maintenance of several machines on a factory. At start, all machines are working and have different probabilities of breaking. The robot must perform maintenance on all machines and avoid the interruption of production. From this environment, two domains were built: *Simple Factory Robot Domain with 3 machines* (SF3) and *Advanced Factory Robot Domain with 3 machines* (AF3). For SF3, 10 different problems were built with different probabilities values. For AF3, 8 different problems were built.

The chosen metrics to compare the dispatchers were: the probability of a successful execution; the average number of replans and actions of successful executions; the average number of actions of failed executions.

Olisipo outperformed the Esterel dispatcher on every metric. It consistently displayed a higher or equal probability of a successful execution and, in addition to this, the mean of the number of replans and of the number of executed actions was consistently lower on the Olisipo algorithm than on the Esterel dispatcher, both for successful and failed executions. From these results, it is possible to conclude that, in comparison to the Esterel dispatcher, the Olisipo algorithm:

- Has a higher probability of solving a problem;
- Needs to replan less times during execution;
- Needs to execute less actions to achieve the goal or before declaring a failure.

The execution of less actions becomes especially relevant in environments where the execution of actions has a cost or results in negative consequences. Considering the factory robot environment used in this thesis, a lower number of executed actions results in slower battery drainage. Hence, the same robot would be able to function for longer periods of time. In addition to this, when the goal is not achieved, it executes less actions and wastes less energy, therefore prolonging even more the robot's work time. This characteristic is also important in environments where the robot cannot be easily charged or accessed, such as space robots.

The smaller number of replans becomes especially relevant in robots which possess limited computation capabilities. In the situation where the factory robot has limited computation capabilities and needs to communicate with a main server for bigger calculations, the Olisipo algorithm allows this robot to perform its function with greater independence and to decrease the number of communications. The reduction in the number of communications prolongs the work time of the robot. In addition to this, the smaller number of replans also enables the use of a simpler computer on the robot, which may also contribute to slower battery drainage.

From these results, it is possible to conclude that Olisipo offers a substantial improvement in performance over a simpler dispatcher, such as the Esterel dispatcher. It consistently executes less actions and needs to replan less times. In the case where a replan is needed, the previously calculated partially-ordered plan and the B&B search contribute to reducing the computation time of replanning.

Something worth referring are the limitations of the Olisipo algorithm. The Olisipo algorithm uses a dynamic execution to handle perturbations in the state of the world. However, if it is not possible to recover from these perturbations by executing the actions included in the plan $\prod'$, then Olisipo will not be able to build a plan Γ which achieves the goal. In addition to this, if a perturbation, which is not modelled by the domain, occurs, then Olisipo will not be able to handle it. Olisipo's ability to recover from perturbations is directly linked to the used domain model and to the action nodes included in $\prod'$.

The source code of the work developed in this thesis is public[1]. Olisipo works as an add-on to any external planner and can easily be implemented by third-parties to improve execution performance. This thesis resulted from an international collaboration between the author of this thesis, Rodrigo Ventura (IST), Oscar Lima (DFKI Robotics Innovation Center), Andrea Micheli (Fondazione Bruno Kessler) and Michael Cashmore (KCL). In addition to this, the work developed in this thesis produced a conference paper to be submitted for ICAPS 2021.

5.1 Future Work

Firstly, let us consider the B&B search. After a plan is selected from the search tree, it is possible that, after executing one of the actions of that plan, another plan with a higher success probability has become applicable, due to unexpected perturbations in the state of the world. In the algorithm developed in this thesis, that possibility is not addressed since it would require saving several totally-ordered plans (demanding higher memory resources) and delay the execution of the plan (since we would need to check if a plan with a higher success probability has become applicable, after each action execution). However, this might be an interesting problem to try and solve in a future work, since it can substantially improve the execution's success probability.

Secondly, there is another consideration to be made regarding the B&B search. In the B&B search, only the nodes in the partially-ordered plan $\prod'$ are considered. The partially-ordered plan $\prod'$ only contains nodes present in the totally-ordered plan $\prod^{tt}$. Hence, if a useful or essential action exists in the domain but it was not present in the plan $\prod^{tt}$, then it will never be considered in the online phase of Olisipo. A simple example of this would be the Factory Robot Domain with machines $m1$ and $m2$ working but not maintained in the initial state, while machine $m3$ is working and maintained. This means that the plan $\prod^{tt}$, which does not account for state perturbations, will only consist of maintaining machines $m1$ and $m2$. If machine $m3$ unexpectedly becomes unmaintained, then the problem becomes unsolvable for Olisipo. However, a simple dispatcher, such as Esterel, would build a new plan and perform maintenance in $m3$. As it can be seen, this is a very limiting feature of Olisipo. This problem could be tackled as a future work and, a very simple solution to it, would be repeating the offline phase after a certain number of plans Γ are consecutively built. Plus, the offline phase should also be repeated if no plan Γ was found during the B&B search.

Thirdly, we have considered that, when the plan Γ stops being compatible with the state of the world, Olisipo needs to repeat PGen from scratch. However, it might be useful to save the previously constructed search tree, as well as its corresponding DBN, and use it to simplify the search for a new Γ. Every time we are using the B&B to search for a new Γ, a part of the search tree that we are building might be equal to another part of a previously built search tree. Hence, it would be interesting to divide the search tree into several different parts and save them for a later use. Along with this, the saving of the DBN corresponding

[1]https://github.com/TomasRibeiro96/Olisipo-planner

to these parts might also contribute to accelerate the building of the DBN and the calculation of the plan's success probability. As an example to illustrate this reasoning, let us consider that a search tree has been built and we decide to save the states and actions between S_2 and the goal state. When we are searching for a new Γ, if the state of the world is compatible with S_2 or if we obtain an expected state compatible with S_2, we can directly add the previously saved states and actions to the new search tree. Hence, this would accelerate the process of searching for a new plan and, in the best case scenario, the search problem could become an ordering of different previously saved search tree parts instead of an ordering of applicable actions.

Fourthly, the fact that the Olisipo algorithm consistently had zero replans on successful executions, indicates that repeating the PGen sub-phase rarely resulted in a successful execution. Hence, it would be interesting the further study the relevance of the PGen sub-phase. From this study, it would be possible to conclude whether the PGen sub-phase should be discarded, improved or substituted by repeating the offline phase, for example.

Fifthly, case-based reasoning could be used to simplify the process of replanning. If this was to be implemented, then if the robot needed to replan from a state of the world that it had already encountered before, then it could simply use the plan it had previously built and avoid the process of replanning.

Sixthly, another improvement that can be made consists in optimising the code itself, to reduce computation time. While developing Olisipo, ROS Services were used to handle communication between scripts. This was used with the purpose of keeping the code simple and easy to read, so that others can make use of it and to accelerate the development of the algorithm itself. However, if we choose to prioritise computation time, there are other more time efficient tools that can be used. For example, Cython [3] could be used to compile the Python code (which builds the DBN) into C++ code and then be called from the C++ code of the B&B search. In addition to this, if time efficiency is to be further improved, then the entire architecture and coordination of the several different scripts could be reviewed.

Seventh and lastly, it would be an interesting contribution to test the Olisipo algorithm in a real world environment, with a real robot. For this thesis, Olisipo was only tested in a simulation environment. However, it is important to understand the usefulness of Olisipo in the real world. How hard is it to probabilistically define the perturbations present in the real world? How does one define the probability of a given action's execution being successful? Would one have to force the robot to repeat a certain action 100 times and determine its success rate? Is this feasible? These are just some of many pertinent questions whose answers dictate the practicality and advantage obtained from the use of Olisipo in real life.

References

[1] A. Cesta, G. Cortellessa, S. F. A. O. and Rasconi, R. (2009). The apsi framework: a planning and scheduling software development environment. *ICAPS 2009*. 17

[2] Baker, M. and Boult, T. (1990). Pruning bayesian networks for efficient computation. In *Sixth Annual Conference on Uncertainty in Artificial Intelligence*, pages 225–232. 18, 19

[3] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D., and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39. VII, 58

[4] Bishop, C. (2006). *Pattern recognition and machine learning*. Springer, New York. 20

[5] Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):281 – 300. 12

[6] Bäckström, C. (1998). Computational aspects of reordering plans. *Journal of Artificial Intelligence Research*, 9:99–137. 17

[7] Casella, G., Robert, C. P., and Wells, M. T. (2004). *Generalized Accept-Reject sampling schemes*, volume 45 of *Lecture Notes–Monograph Series*, pages 342–347. Institute of Mathematical Statistics, Beachwood, Ohio, USA. 20

[8] Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., Carrera, A., Palomeras, N., Hurtós, N., and Carreras, M. (2015). Rosplan: Planning in the robot operating system. *Proceedings International Conference on Automated Planning and Scheduling, ICAPS*, 2015:333–341. III, VI, XI, 21, 22, 43, 44, 55

[9] Cimatti, A., Micheli, A., and Roveri, M. (2016). Dynamic controllability of disjunctive temporal networks: Validation and synthesis of executable strategies. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 3116–3122. AAAI Press. 18

[10] Coles, A., Coles, A., Fox, M., and Long, D. (2010). Forward-chaining partial-order planning. *ICAPS 2010 - Proceedings of the 20th International Conference on Automated Planning and Scheduling*, pages 42–49. VI, 22, 43, 44

[11] Dagum, P., Galper, A., and Horvitz, E. (1992). Dynamic network models for forecasting. In *Uncertainty in Artificial Intelligence*, pages 41 – 48. Morgan Kaufmann. 29

[12] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397. 12

[13] Dechter, R., Meiri, I., and Pearl, J. (1991). Temporal constraint networks. *Artificial Intelligence*, 49(1):61–95. 18

[14] Do, M. and Kambhampati, S. (2003). Improving temporal flexibility of position constrained metric temporal plans. In *Proceeding of ICAPS 2003*, pages 42–51. 17

[15] Edelkamp, S. and Hoffmann, J. (2004). Pddl2. 2: The language for the classical part of the 4th international planning competition. *ICAPS 2004*. 15

[16] Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189 – 208. 7, 15

[17] Fox, M. and Long, D. (2003). Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124. 15, 17

[18] Frank, J. and Jonsson, A. (2003). Constraint-based attribute and interval planning. *Constraints*, 8:339–364. 17, 18

[19] Frank, J. and Morris, P. H. (2007). Bounding the resource availability of activities with linear resource impact. *International Conference on Automated Planning and Scheduling 2007*, pages 136–143. 17

[20] Geman, S. and Geman, D. (1984). Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741. 20

[21] Gerevini, A. and Long, D. (2005). Plan constraints and preferences in pddl3 the language of the fifth international planning competition. *ICAPS 2006*. 15

[22] Ghallab, M., Knoblock, C., Wilkins, D., Barrett, A., Christianson, D., Friedman, M., Kwok, C., Golden, K., Penberthy, S., Smith, D., Sun, Y., and Weld, D. (1998). Pddl - the planning domain definition language. *AIPS 1998*. 8, 15

[23] Ghallab, M. and Laruelle, H. (1994). Representation and control in ixtet, a temporal planner. In *Proceedings of AIPS*, pages 61–67. 17

[24] Ghallab, M., Nau, D., and Traverso, P. (2016). *Automated Planning and Acting*. Cambridge University Press. 8

[25] Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107. 10

[26] Kim, P., Williams, B. C., and Abramson, M. (2001). Executing reactive, model-based programs through graph-based temporal planning. *International Joint Conferences on Artificial Intelligence*, pages 487–493. 18

[27] Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press. 20

[28] Kovacs, D. L. (2011). Bnf definition of pddl 3.1. *Unpublished manuscript from the IPC-2011 website*. 15

[29] Land, A. H. and Doig, A. G. (2010). *An Automatic Method for Solving Discrete Programming Problems*, pages 105–132. Springer Berlin Heidelberg, Berlin, Heidelberg. V, 2, 11, 18

[30] Lee, C. Y. (1961). An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365. 10

[31] Levine, S. J. and Williams, B. C. (2018). Watching and acting together: Concurrent plan recognition and adaptation for human-robot teams. *Journal of Artificial Intelligence Research*, 63:281–359. 18

[32] Lima, O., Ventura, R., and Awaad, I. (2018). Integrating classical planning and real robots in industrial and service robotics domains. *ICAPS 2018 Workshop*. 2

[33] Lima Carrion, O., Cashmore, M., Magazzeni, D., Micheli, A., and Ventura, R. (2020). Robust plan execution with unexpected observations. *Pre-print*. [2]. III, V, VI, XI, XII, 2, 18, 23, 25, 26, 27, 43, 55

[34] McCulloch, W. S. and Pitts, W. (1990). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 52(1):99 – 115. 5

[35] Meekes, M., Renooij, S., and van der Gaag, L. C. (2015). Relevance of evidence in bayesian networks. *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 366–375. 19

[36] Moore, E. (1959). *The Shortest Path Through a Maze*. Bell Telephone System. Technical publications. monograph. Bell Telephone System. 10

[37] Neal, R. M. (2003). Slice sampling. *Ann. Statist.*, 31(3):705–767. 20

[38] Newell, A., Shaw, J., and Simon, H. (1959). Report on a general problem-solving program. 7

[2]https://www.researchgate.net/publication/340094738_Robust_Plan_Execution_with_Unexpected_Observations

[39] Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. *ICRA Workshop on Open Source Software*. 21

[40] Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition. XI, 6, 9, 11, 12, 14, 15, 19, 20, 21, 29, 38

[41] Selman, B., Kautz, H., and Cohen, B. (1999). Local search strategies for satisfiability testing. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, 26. 12

[42] Turing, A. (2004). *The Essential Turing: The ideas that gave birth to the computer age, p. 412*. Oxford University Press. V, 5

[43] Umbrico, A., Cesta, A., Mayer, M. C., and Orlandini, A. (2018). Integrating resource management and timeline-based planning. *International Conference on Automated Planning and Scheduling 2018*, pages 264–272. 17

[44] Wilson, E. B. (1927). Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212. 46

[45] Zeng, W. and Church, R. (2009). Finding shortest paths on real road networks: The case for a*. *International Journal of Geographical Information Science*, 23:531–543. 10